# systemd для администраторов

Lennart Poettering (автор)*
Сергей Пташник (русский перевод)†


25 января 2013 г.

## Содержание



*Первоисточник (на английском языке) опубликован на сайте автора: http://0pointer.de/blog/projects

†Актуальная версия перевода доступна на личной странице переводчика: http://www2.kangran.su/~nnz/pub/s4a/

# Предисловие автора

Многие из вас, наверное, уже знают, что systemd — это новая система инициализации дистрибутива Fedora, начиная с Fedora 14[1]. Помимо Fedora, systemd также поддерживает и другие дистрибутивы, в частности, OpenSUSE[2]. systemd предоставляет администраторам целый ряд новых возможностей, значительно упрощающих процесс обслуживания системы. Эта статья является первой в серии публикаций, планируемых в ближайшие месяцы. В каждой из этих статей я попытаюсь рассказать об очередной новой возможности systemd. Большинство этих возможностей можно описать легко и просто, и подобные статьи должны быть интересны довольно широкой аудитории. Однако, время от времени мы будем рассматривать ключевые новшества systemd, что может потребовать несколько более подробного изложения.

Lennart Poettering, 23 августа 2010 г.

# 1 Контроль процесса загрузки

Как правило, во время загрузки Linux по экрану быстро пробегает огромное количество различных сообщений. Так как мы интенсивно работаем над параллелизацией и ускорением процесса загрузки, с каждой новой версией systemd эти сообщения будут пробегать все быстрее и быстрее, вследствие чего, читать их будет все труднее. К тому же, многие пользователи применяют графические оболочки загрузки (например,

[1] Прим. перев.: к сожалению, разработчики Fedora приняли решение оставить в Fedora 14 в качестве системы инициализации по умолчанию upstart, однако systemd все равно включен в этот релиз и может быть использован в качестве альтернативной системы инициализации. Окончательный переход на systemd произошел лишь в Fedora 15.

[2] Прим. перев.: Сейчас systemd поддерживается практически во всех популярных дистрибутивах для настольных систем.

Plymouth), полностью скрывающие эти сообщения. Тем не менее, информация, которую они несут, была и остается чрезвычайно важной — они показывают, успешно ли запустилась каждая служба, или попытка ее запуска закончилась ошибкой (зеленое `[ OK ]` или красное `[ FAILED ]` соответственно). Итак, с ростом скорости загрузки систем, возникает неприятная ситуация: информация о результатах запуска служб бывает очень важна, а просматривать ее все тяжелее. systemd предлагает выход из этой ситуации: он отслеживает и запоминает факты успешного или неудачного запуска служб на этапе загрузки, а также сбои служб во время работы. К таким случаям относятся выходы с ненулевым кодом, ошибки сегментирования и т.п. Введя `systemctl status` в своей командной оболочке, вы можете ознакомиться с состоянием всех служб, как «родных» (native) для systemd, так и классических SysV/LSB служб, поддерживаемых в целях совместимости:

```
[root@lambda] ~# systemctl
UNIT                                        LOAD   ACTIVE       SUB        JOB       DESCRIPTION
dev-hugepages.automount                     loaded active       running              Huge Pages File System Automount Point
dev-mqueue.automount                        loaded active       running              POSIX Message Queue File System Automount Point
proc-sys-fs-binfmt_misc.automount           loaded active       waiting              Arbitrary Executable File Formats File System Automount Point
sys-kernel-debug.automount                  loaded active       waiting              Debug File System Automount Point
sys-kernel-security.automount               loaded active       waiting              Security File System Automount Point
sys-devices-pc...0000:02:00.0-net-eth0.device loaded active     plugged              82573L Gigabit Ethernet Controller
[...]
sys-devices-virtual-tty-tty9.device         loaded active       plugged              /sys/devices/virtual/tty/tty9
-.mount                                     loaded active       mounted              /
boot.mount                                  loaded active       mounted              /boot
dev-hugepages.mount                         loaded active       mounted              Huge Pages File System
dev-mqueue.mount                            loaded active       mounted              POSIX Message Queue File System
home.mount                                  loaded active       mounted              /home
proc-sys-fs-binfmt_misc.mount               loaded active       mounted              Arbitrary Executable File Formats File System
abrtd.service                               loaded active       running              ABRT Automated Bug Reporting Tool
accounts-daemon.service                     loaded active       running              Accounts Service
acpid.service                               loaded active       running              ACPI Event Daemon
atd.service                                 loaded active       running              Execution Queue Daemon
auditd.service                              loaded active       running              Security Auditing Service
avahi-daemon.service                        loaded active       running              Avahi mDNS/DNS-SD Stack
bluetooth.service                           loaded active       running              Bluetooth Manager
console-kit-daemon.service                  loaded active       running              Console Manager
cpuspeed.service                            loaded active       exited               LSB: processor frequency scaling support
crond.service                               loaded active       running              Command Scheduler
cups.service                                loaded active       running              CUPS Printing Service
dbus.service                                loaded active       running              D-Bus System Message Bus
getty@tty2.service                          loaded active       running              Getty on tty2
getty@tty3.service                          loaded active       running              Getty on tty3
getty@tty4.service                          loaded active       running              Getty on tty4
getty@tty5.service                          loaded active       running              Getty on tty5
getty@tty6.service                          loaded active       running              Getty on tty6
haldaemon.service                           loaded active       running              Hardware Manager
hdapsd@sda.service                          loaded active       running              sda shock protection daemon
irqbalance.service                          loaded active       running              LSB: start and stop irqbalance daemon
```

```
iscsi.service                                    loaded active       exited      LSB: Starts and stops login and scanning of iSCSI devices.
iscsid.service                                   loaded active       exited      LSB: Starts and stops login iSCSI daemon.
livesys-late.service                             loaded active       exited      LSB: Late init script for live image.
livesys.service                                  loaded active       exited      LSB: Init script for live image.
lvm2-monitor.service                             loaded active       exited      LSB: Monitoring of LVM2 mirrors, snapshots etc. using dmeventd or progr
mdmonitor.service                                loaded active       running     LSB: Start and stop the MD software RAID monitor
modem-manager.service                            loaded active       running     Modem Manager
netfs.service                                    loaded active       exited      LSB: Mount and unmount network filesystems.
NetworkManager.service                           loaded active       running     Network Manager
ntpd.service                                     loaded maintenance  maintenance Network Time Service
polkitd.service                                  loaded active       running     Policy Manager
prefdm.service                                   loaded active       running     Display Manager
rc-local.service                                 loaded active       exited      /etc/rc.local Compatibility
rpcbind.service                                  loaded active       running     RPC Portmapper Service
rsyslog.service                                  loaded active       running     System Logging Service
rtkit-daemon.service                             loaded active       running     RealtimeKit Scheduling Policy Service
sendmail.service                                 loaded active       running     LSB: start and stop sendmail
sshd@172.31.0.53:22-172.31.0.4:36368.service     loaded active       running     SSH Per-Connection Server
sysinit.service                                  loaded active       running     System Initialization
systemd-logger.service                           loaded active       running     systemd Logging Daemon
udev-post.service                                loaded active       exited      LSB: Moves the generated persistent udev rules to /etc/udev/rules.d
udisks.service                                   loaded active       running     Disk Manager
upowerd.service                                  loaded active       running     Power Manager
wpa_supplicant.service                           loaded active       running     Wi-Fi Security Service
avahi-daemon.socket                              loaded active       listening   Avahi mDNS/DNS-SD Stack Activation Socket
cups.socket                                      loaded active       listening   CUPS Printing Service Sockets
dbus.socket                                      loaded active       running     dbus.socket
rpcbind.socket                                   loaded active       listening   RPC Portmapper Socket
sshd.socket                                      loaded active       listening   sshd.socket
systemd-initctl.socket                           loaded active       listening   systemd /dev/initctl Compatibility Socket
systemd-logger.socket                            loaded active       running     systemd Logging Socket
systemd-shutdownd.socket                         loaded active       listening   systemd Delayed Shutdown Socket
dev-disk-by\x1...x1db22a\x1d870f1adf2732.swap    loaded active       active      /dev/disk/by-uuid/fd626ef7-34a4-4958-b22a-870f1adf2732
basic.target                                     loaded active       active      Basic System
bluetooth.target                                 loaded active       active      Bluetooth
dbus.target                                      loaded active       active      D-Bus
```

```
getty.target                                loaded active       active              Login Prompts
graphical.target                            loaded active       active              Graphical Interface
local-fs.target                             loaded active       active              Local File Systems
multi-user.target                           loaded active       active              Multi-User
network.target                              loaded active       active              Network
remote-fs.target                            loaded active       active              Remote File Systems
sockets.target                              loaded active       active              Sockets
swap.target                                 loaded active       active              Swap
sysinit.target                              loaded active       active              System Initialization

LOAD   = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB    = The low-level unit activation state, values depend on unit type.
JOB    = Pending job for the unit.

221 units listed. Pass --all to see inactive units, too.
[root@lambda] ~#
```



(Листинг был сокращен за счет удаления строк, не относящихся к теме статьи.)

Обратите внимание на графу ACTIVE, в которой отображается обобщенный статус службы (или любого другого юнита systemd: устройства, сокета, точки монтирования — их мы рассмотрим подробнее в последующих статьях). Основными значениями обобщенного статуса являются active (служба выполняется) и inactive (служба не была запущена). Также существуют и другие статусы. Например, внимательно посмотрев на листинг выше, вы можете заметить, что служба ntpd (сервер точного времени) находится в состоянии, обозначенном как maintenance. Чтобы узнать, что же произошло с ntpd, воспользуемся командой `systemctl status`[3]:

```
[root@lambda] ~# systemctl status ntpd.service
ntpd.service - Network Time Service
          Loaded: loaded (/etc/systemd/system/ntpd.service)
          Active: maintenance
            Main: 953 (code=exited, status=255)
          CGroup: name=systemd:/systemd-1/ntpd.service
[root@lambda] ~#
```

systemd сообщает нам, что ntpd был запущен (с идентификатором процесса 953) и аварийно завершил работу (с кодом выхода 255)[4].

В последующих версиях systemd, мы планируем добавить возможность вызова в таких ситуациях ABRT (Automated Bug Report Tool), но для этого необходима поддержка со стороны самого ABRT. Соответствующий запрос уже направлен его разработчикам, однако пока не встретил среди них поддержки.

Резюме: использование `systemctl` и `systemctl status` является современной, более удобной и эффективной альтернативой разглядыванию быстро пробегающих по экрану сообщений в классическом SysV. `systemctl status` дает возможность получить развернутую информацию о характере ошибки и, кроме того, в отличие от сообщений SysV, показывает не только ошибки при запуске, но и ошибки, возникшие во время исполнения службы.

## 2 О службах и процессах

В большинстве современных Linux-систем количество одновременно работающих процессов обычно весьма значительно. Понять, откуда взялся и что делает тот или иной процесс, становится все сложнее и сложнее. Многие службы используют сразу несколько рабочих процессов, и это отнюдь не всегда можно легко распознать по выводу команды `ps`. Встречаются еще более сложные ситуации, когда демон запускает сторонние процессы — например, веб-сервер выполняет CGI-программы, а демон cron — команды, предписанные ему в crontab.

Немного помочь в решении этой проблемы может древовидная иерархия процессов, отображаемая по команде `ps xaf`. Именно «немного помочь», а не решить полностью. В частности, процессы, родители которых умирают раньше их самих, становят потомками PID 1 (процесса init), что сразу затрудняет процесс выяснения их происхождения. Кроме того, процесс может избавиться от связи с родителем через две последовательные операции `fork()` (в целом, эта возможность признается нужной и полезной, и является частью используемого в Unix подхода к разработке демонов). Также, не будем забывать, что процесс легко может изменить свое имя посредством `PR_SETNAME`, или задав значение `argv[0]`, что также усложняет процесс его опознания[5].

[3] Прим. перев.: Стоит заметить, что формат вывода данной команды менялся по мере развития systemd — появлялись дополнительные поля с информацией, был добавлен вывод журнала службы (см. главу 13) и т.д. Здесь приведен пример вывода этой команды на момент написания исходной статьи (лето 2010 года).

[4] Прим. перев.: Впоследствии, про просьбам пользователей, считавших, что слово «maintenance» недостаточно точно отражает ситуацию, оно было заменено на «failed».

[5] Прим. перев.: Стоит отметить, что перечисленные ситуации могут возникнуть не только вследствие ошибок в коде и/или конфигурации программ, но и в результате злого умысла. Например, очень часто встречается ситуация, когда установленный на взломанном сервере процесс-бэкдор маскируется под нормального демона, меняя себе имя, скажем, на httpd.

systemd предлагает простой путь для решения обсуждаемой задачи. Запуская новый процесс, systemd помещает его в отдельную контрольную группу с соответствующим именем. Контрольные группы Linux предоставляют очень удобный инструмент для иерархической структуризации процессов: когда какой-либо процесс порождает потомка, этот потомок автоматически включается в ту же группу, что и родитель. При этом, что очень важно, непривилегированные процессы не могут изменить свое положение в этой иерархии. Таким образом, контрольные группы позволяют точно установить происхождение конкретного процесса, вне зависимости от того, сколько раз он форкался и переименовывал себя — имя его контрольной группы невозможно спрятать или изменить. Кроме того, при штатном завершении родительской службы, будут завершены и все порожденные ею процессы, как бы они ни пытались сбежать. С systemd уже невозможна ситуация, когда после остановки web-сервера, некорректно форкнувшийся CGI-процесс продолжает исполняться вплоть до последних секунд работы системы.

В этой статье мы рассмотрим две простых команды, которые позволят вам наглядно оценить схему взаимоотношений systemd и порожденных им процессов. Первая из этих команд — все та же `ps`, однако на этот раз в ее параметры добавлено указание выводить сведения по контрольным группам, а также другую интересную информацию:

```
$ ps xawf -eo pid,user,cgroup,args
  PID USER     CGROUP                              COMMAND
    2 root     -                                   [kthreadd]
    3 root     -                                    \_ [ksoftirqd/0]
[...]
 4281 root     -                                    \_ [flush-8:0]
    1 root     name=systemd:/systemd-1             /sbin/init
  455 root     name=systemd:/systemd-1/sysinit.service /sbin/udevd -d
28188 root     name=systemd:/systemd-1/sysinit.service  \_ /sbin/udevd -d
28191 root     name=systemd:/systemd-1/sysinit.service  \_ /sbin/udevd -d
 1096 dbus     name=systemd:/systemd-1/dbus.service /bin/dbus-daemon --system --address=systemd: --nofork --systemd-activation
 1131 root     name=systemd:/systemd-1/auditd.service auditd
 1133 root     name=systemd:/systemd-1/auditd.service  \_ /sbin/audispd
 1135 root     name=systemd:/systemd-1/auditd.service      \_ /usr/sbin/sedispatch
 1171 root     name=systemd:/systemd-1/NetworkManager.service /usr/sbin/NetworkManager --no-daemon
 4028 root     name=systemd:/systemd-1/NetworkManager.service  \_ /sbin/dhclient -d -4 -sf /usr/libexec/nm-dhcp-client.action -pf /var/run/dhclient-wlan0.pid -
 1175 avahi    name=systemd:/systemd-1/avahi-daemon.service avahi-daemon: running [epsilon.local]
 1194 avahi    name=systemd:/systemd-1/avahi-daemon.service  \_ avahi-daemon: chroot helper
 1193 root     name=systemd:/systemd-1/rsyslog.service /sbin/rsyslogd -c 4
 1195 root     name=systemd:/systemd-1/cups.service cupsd -C /etc/cups/cupsd.conf
 1207 root     name=systemd:/systemd-1/mdmonitor.service mdadm --monitor --scan -f --pid-file=/var/run/mdadm/mdadm.pid
 1210 root     name=systemd:/systemd-1/irqbalance.service irqbalance
 1216 root     name=systemd:/systemd-1/dbus.service /usr/sbin/modem-manager
 1219 root     name=systemd:/systemd-1/dbus.service /usr/libexec/polkit-1/polkitd
 1242 root     name=systemd:/systemd-1/dbus.service /usr/sbin/wpa_supplicant -c /etc/wpa_supplicant/wpa_supplicant.conf -B -u -f /var/log/wpa_supplicant.log -I
 1249 68       name=systemd:/systemd-1/haldaemon.service hald
 1250 root     name=systemd:/systemd-1/haldaemon.service  \_ hald-runner
 1273 root     name=systemd:/systemd-1/haldaemon.service      \_ hald-addon-input: Listening on /dev/input/event3 /dev/input/event9 /dev/input/event1 /dev/inpu
 1275 root     name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-rfkill-killswitch
 1284 root     name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-leds
 1285 root     name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-generic-backlight
 1287 68       name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-acpi
 1317 root     name=systemd:/systemd-1/abrtd.service /usr/sbin/abrtd -d -s
 1332 root     name=systemd:/systemd-1/getty@.service/tty2 /sbin/mingetty tty2
 1339 root     name=systemd:/systemd-1/getty@.service/tty3 /sbin/mingetty tty3
 1342 root     name=systemd:/systemd-1/getty@.service/tty5 /sbin/mingetty tty5
```

```
1343 root     name=systemd:/systemd-1/getty@.service/tty4 /sbin/mingetty tty4
1344 root     name=systemd:/systemd-1/crond.service crond
1346 root     name=systemd:/systemd-1/getty@.service/tty6 /sbin/mingetty tty6
1362 root     name=systemd:/systemd-1/sshd.service /usr/sbin/sshd
1376 root     name=systemd:/systemd-1/prefdm.service /usr/sbin/gdm-binary -nodaemon
1391 root     name=systemd:/systemd-1/prefdm.service  \_ /usr/libexec/gdm-simple-slave --display-id /org/gnome/DisplayManager/Display1 --force-active-vt
1394 root     name=systemd:/systemd-1/prefdm.service      \_ /usr/bin/Xorg :0 -nr -verbose -auth /var/run/gdm/auth-for-gdm-f2KUOh/database -nolisten tcp vt1
1495 root     name=systemd:/user/lennart/1             \_ pam: gdm-password
1521 lennart  name=systemd:/user/lennart/1                 \_ gnome-session
1621 lennart  name=systemd:/user/lennart/1                     \_ metacity
1635 lennart  name=systemd:/user/lennart/1                     \_ gnome-panel
1638 lennart  name=systemd:/user/lennart/1                     \_ nautilus
1640 lennart  name=systemd:/user/lennart/1                     \_ /usr/libexec/polkit-gnome-authentication-agent-1
1641 lennart  name=systemd:/user/lennart/1                     \_ /usr/bin/seapplet
1644 lennart  name=systemd:/user/lennart/1                     \_ gnome-volume-control-applet
1646 lennart  name=systemd:/user/lennart/1                     \_ /usr/sbin/restorecond -u
1652 lennart  name=systemd:/user/lennart/1                     \_ /usr/bin/devilspie
1662 lennart  name=systemd:/user/lennart/1                     \_ nm-applet --sm-disable
1664 lennart  name=systemd:/user/lennart/1                     \_ gnome-power-manager
1665 lennart  name=systemd:/user/lennart/1                     \_ /usr/libexec/gdu-notification-daemon
1670 lennart  name=systemd:/user/lennart/1                     \_ /usr/libexec/evolution/2.32/evolution-alarm-notify
1672 lennart  name=systemd:/user/lennart/1                     \_ /usr/bin/python /usr/share/system-config-printer/applet.py
1674 lennart  name=systemd:/user/lennart/1                     \_ /usr/lib64/deja-dup/deja-dup-monitor
1675 lennart  name=systemd:/user/lennart/1                     \_ abrt-applet
1677 lennart  name=systemd:/user/lennart/1                     \_ bluetooth-applet
1678 lennart  name=systemd:/user/lennart/1                     \_ gpk-update-icon
1408 root     name=systemd:/systemd-1/console-kit-daemon.service /usr/sbin/console-kit-daemon --no-daemon
1419 gdm      name=systemd:/systemd-1/prefdm.service /usr/bin/dbus-launch --exit-with-session
1453 root     name=systemd:/systemd-1/dbus.service /usr/libexec/upowerd
1473 rtkit    name=systemd:/systemd-1/rtkit-daemon.service /usr/libexec/rtkit-daemon
1496 root     name=systemd:/systemd-1/accounts-daemon.service /usr/libexec/accounts-daemon
1499 root     name=systemd:/systemd-1/systemd-logger.service /lib/systemd/systemd-logger
1511 lennart  name=systemd:/systemd-1/prefdm.service /usr/bin/gnome-keyring-daemon --daemonize --login
1534 lennart  name=systemd:/user/lennart/1        dbus-launch --sh-syntax --exit-with-session
1535 lennart  name=systemd:/user/lennart/1        /bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
1603 lennart  name=systemd:/user/lennart/1        /usr/libexec/gconfd-2
```

```
 1612 lennart  name=systemd:/user/lennart/1        /usr/libexec/gnome-settings-daemon
 1615 lennart  name=systemd:/user/lennart/1        /usr/libexec/gvfsd
 1626 lennart  name=systemd:/user/lennart/1        /usr/libexec//gvfs-fuse-daemon /home/lennart/.gvfs
 1634 lennart  name=systemd:/user/lennart/1        /usr/bin/pulseaudio --start --log-target=syslog
 1649 lennart  name=systemd:/user/lennart/1         \_ /usr/libexec/pulse/gconf-helper
 1645 lennart  name=systemd:/user/lennart/1        /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=24
 1668 lennart  name=systemd:/user/lennart/1        /usr/libexec/im-settings-daemon
 1701 lennart  name=systemd:/user/lennart/1        /usr/libexec/gvfs-gdu-volume-monitor
 1707 lennart  name=systemd:/user/lennart/1        /usr/bin/gnote --panel-applet --oaf-activate-iid=OAFIID:GnoteApplet_Factory --oaf-ior-fd=22
 1725 lennart  name=systemd:/user/lennart/1        /usr/libexec/clock-applet
 1727 lennart  name=systemd:/user/lennart/1        /usr/libexec/wnck-applet
 1729 lennart  name=systemd:/user/lennart/1        /usr/libexec/notification-area-applet
 1733 root     name=systemd:/systemd-1/dbus.service /usr/libexec/udisks-daemon
 1747 root     name=systemd:/systemd-1/dbus.service  \_ udisks-daemon: polling /dev/sr0
 1759 lennart  name=systemd:/user/lennart/1        gnome-screensaver
 1780 lennart  name=systemd:/user/lennart/1        /usr/libexec/gvfsd-trash --spawner :1.9 /org/gtk/gvfs/exec_spaw/0
 1864 lennart  name=systemd:/user/lennart/1        /usr/libexec/gvfs-afc-volume-monitor
 1874 lennart  name=systemd:/user/lennart/1        /usr/libexec/gconf-im-settings-daemon
 1903 lennart  name=systemd:/user/lennart/1        /usr/libexec/gvfsd-burn --spawner :1.9 /org/gtk/gvfs/exec_spaw/1
 1909 lennart  name=systemd:/user/lennart/1        gnome-terminal
 1913 lennart  name=systemd:/user/lennart/1         \_ gnome-pty-helper
 1914 lennart  name=systemd:/user/lennart/1         \_ bash
29231 lennart  name=systemd:/user/lennart/1         |   \_ ssh tango
 2221 lennart  name=systemd:/user/lennart/1         \_ bash
 4193 lennart  name=systemd:/user/lennart/1         |   \_ ssh tango
 2461 lennart  name=systemd:/user/lennart/1         \_ bash
29219 lennart  name=systemd:/user/lennart/1         |   \_ emacs systemd-for-admins-1.txt
15113 lennart  name=systemd:/user/lennart/1         \_ bash
27251 lennart  name=systemd:/user/lennart/1             \_ empathy
29504 lennart  name=systemd:/user/lennart/1             \_ ps xawf -eo pid,user,cgroup,args
 1968 lennart  name=systemd:/user/lennart/1        ssh-agent
 1994 lennart  name=systemd:/user/lennart/1        gpg-agent --daemon --write-env-file
18679 lennart  name=systemd:/user/lennart/1        /bin/sh /usr/lib64/firefox-3.6/run-mozilla.sh /usr/lib64/firefox-3.6/firefox
18741 lennart  name=systemd:/user/lennart/1         \_ /usr/lib64/firefox-3.6/firefox
28900 lennart  name=systemd:/user/lennart/1             \_ /usr/lib64/nspluginwrapper/npviewer.bin --plugin /usr/lib64/mozilla/plugins/libflashplayer.so --con
 4016 root     name=systemd:/systemd-1/sysinit.service /usr/sbin/bluetoothd --udev
```

```
 4094 smmsp    name=systemd:/systemd-1/sendmail.service sendmail: Queue runner@01:00:00 for /var/spool/clientmqueue
 4096 root     name=systemd:/systemd-1/sendmail.service sendmail: accepting connections
 4112 ntp      name=systemd:/systemd-1/ntpd.service /usr/sbin/ntpd -n -u ntp:ntp -g
27262 lennart  name=systemd:/user/lennart/1        /usr/libexec/mission-control-5
27265 lennart  name=systemd:/user/lennart/1        /usr/libexec/telepathy-haze
27268 lennart  name=systemd:/user/lennart/1        /usr/libexec/telepathy-logger
27270 lennart  name=systemd:/user/lennart/1        /usr/libexec/dconf-service
27280 lennart  name=systemd:/user/lennart/1        /usr/libexec/notification-daemon
27284 lennart  name=systemd:/user/lennart/1        /usr/libexec/telepathy-gabble
27285 lennart  name=systemd:/user/lennart/1        /usr/libexec/telepathy-salut
27297 lennart  name=systemd:/user/lennart/1        /usr/libexec/geoclue-yahoo
```

(Данный листинг был сокращен за счет удаления из него строк, описывающих потоки ядра, так как они никак не относятся к обсуждаемой нами теме.)

Обратите внимание на третий столбец, показывающий имя контрольной группы, в которую systemd поместил данный процесс. Например, процесс `udev` находится в группе `name=systemd:/systemd-1/sysinit.service`. В эту группу входят процессы, порожденные службой `sysinit.service`, которая запускается на ранней стадии загрузки.

Вы можете очень сильно упростить себе работу, если назначите для вышеприведенной команды какой-нибудь простой и короткий псевдоним, например

```
alias psc='ps xawf -eo pid,user,cgroup,args'
```

— теперь для получения исчерпывающей информации по процессам достаточно будет нажать всего четыре клавиши.

Альтернативный способ получить ту же информацию — воспользоваться утилитой `systemd-cgls`, входящей в комплект поставки systemd. Она отображает иерархию контрольных групп в виде псевдографической диаграммы-дерева:

```
$ systemd-cgls
+    2 [kthreadd]
[...]
+ 4281 [flush-8:0]
+ user
| \ lennart
|   \ 1
|     +  1495 pam: gdm-password
|     +  1521 gnome-session
|     +  1534 dbus-launch --sh-syntax --exit-with-session
|     +  1535 /bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
|     +  1603 /usr/libexec/gconfd-2
|     +  1612 /usr/libexec/gnome-settings-daemon
|     +  1615 /ushr/libexec/gvfsd
|     +  1621 metacity
|     +  1626 /usr/libexec//gvfs-fuse-daemon /home/lennart/.gvfs
|     +  1634 /usr/bin/pulseaudio --start --log-target=syslog
|     +  1635 gnome-panel
|     +  1638 nautilus
|     +  1640 /usr/libexec/polkit-gnome-authentication-agent-1
|     +  1641 /usr/bin/seapplet
|     +  1644 gnome-volume-control-applet
|     +  1645 /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=24
|     +  1646 /usr/sbin/restorecond -u
|     +  1649 /usr/libexec/pulse/gconf-helper
|     +  1652 /usr/bin/devilspie
|     +  1662 nm-applet --sm-disable
|     +  1664 gnome-power-manager
|     +  1665 /usr/libexec/gdu-notification-daemon
|     +  1668 /usr/libexec/im-settings-daemon
|     +  1670 /usr/libexec/evolution/2.32/evolution-alarm-notify
|     +  1672 /usr/bin/python /usr/share/system-config-printer/applet.py
|     +  1674 /usr/lib64/deja-dup/deja-dup-monitor
|     +  1675 abrt-applet
|     +  1677 bluetooth-applet
|     +  1678 gpk-update-icon
```

```
|     +  1701 /usr/libexec/gvfs-gdu-volume-monitor
|     +  1707 /usr/bin/gnote --panel-applet --oaf-activate-iid=OAFIID:GnoteApplet_Factory --oaf-ior-fd=22
|     +  1725 /usr/libexec/clock-applet
|     +  1727 /usr/libexec/wnck-applet
|     +  1729 /usr/libexec/notification-area-applet
|     +  1759 gnome-screensaver
|     +  1780 /usr/libexec/gvfsd-trash --spawner :1.9 /org/gtk/gvfs/exec_spaw/0
|     +  1864 /usr/libexec/gvfs-afc-volume-monitor
|     +  1874 /usr/libexec/gconf-im-settings-daemon
|     +  1882 /usr/libexec/gvfs-gphoto2-volume-monitor
|     +  1903 /usr/libexec/gvfsd-burn --spawner :1.9 /org/gtk/gvfs/exec_spaw/1
|     +  1909 gnome-terminal
|     +  1913 gnome-pty-helper
|     +  1914 bash
|     +  1968 ssh-agent
|     +  1994 gpg-agent --daemon --write-env-file
|     +  2221 bash
|     +  2461 bash
|     +  4193 ssh tango
|     + 15113 bash
|     + 18679 /bin/sh /usr/lib64/firefox-3.6/run-mozilla.sh /usr/lib64/firefox-3.6/firefox
|     + 18741 /usr/lib64/firefox-3.6/firefox
|     + 27251 empathy
|     + 27262 /usr/libexec/mission-control-5
|     + 27265 /usr/libexec/telepathy-haze
|     + 27268 /usr/libexec/telepathy-logger
|     + 27270 /usr/libexec/dconf-service
|     + 27280 /usr/libexec/notification-daemon
|     + 27284 /usr/libexec/telepathy-gabble
|     + 27285 /usr/libexec/telepathy-salut
|     + 27297 /usr/libexec/geoclue-yahoo
|     + 28900 /usr/lib64/nspluginwrapper/npviewer.bin --plugin /usr/lib64/mozilla/plugins/libflashplayer.so --connection /org/wrapper/NSPlugins/libflashplayer
|     + 29219 emacs systemd-for-admins-1.txt
|     + 29231 ssh tango
|     \ 29519 systemd-cgls
\ systemd-1
```

```
+ 1 /sbin/init
+ ntpd.service
| \ 4112 /usr/sbin/ntpd -n -u ntp:ntp -g
+ systemd-logger.service
| \ 1499 /lib/systemd/systemd-logger
+ accounts-daemon.service
| \ 1496 /usr/libexec/accounts-daemon
+ rtkit-daemon.service
| \ 1473 /usr/libexec/rtkit-daemon
+ console-kit-daemon.service
| \ 1408 /usr/sbin/console-kit-daemon --no-daemon
+ prefdm.service
| + 1376 /usr/sbin/gdm-binary -nodaemon
| + 1391 /usr/libexec/gdm-simple-slave --display-id /org/gnome/DisplayManager/Display1 --force-active-vt
| + 1394 /usr/bin/Xorg :0 -nr -verbose -auth /var/run/gdm/auth-for-gdm-f2KUOh/database -nolisten tcp vt1
| + 1419 /usr/bin/dbus-launch --exit-with-session
| \ 1511 /usr/bin/gnome-keyring-daemon --daemonize --login
+ getty@.service
| + tty6
| | \ 1346 /sbin/mingetty tty6
| + tty4
| | \ 1343 /sbin/mingetty tty4
| + tty5
| | \ 1342 /sbin/mingetty tty5
| + tty3
| | \ 1339 /sbin/mingetty tty3
| \ tty2
|   \ 1332 /sbin/mingetty tty2
+ abrtd.service
| \ 1317 /usr/sbin/abrtd -d -s
+ crond.service
| \ 1344 crond
+ sshd.service
| \ 1362 /usr/sbin/sshd
+ sendmail.service
| + 4094 sendmail: Queue runner@01:00:00 for /var/spool/clientmqueue
```

```
| \ 4096 sendmail: accepting connections
+ haldaemon.service
| + 1249 hald
| + 1250 hald-runner
| + 1273 hald-addon-input: Listening on /dev/input/event3 /dev/input/event9 /dev/input/event1 /dev/input/event7 /dev/input/event2 /dev/input/event0 /dev/inpu
| + 1275 /usr/libexec/hald-addon-rfkill-killswitch
| + 1284 /usr/libexec/hald-addon-leds
| + 1285 /usr/libexec/hald-addon-generic-backlight
| \ 1287 /usr/libexec/hald-addon-acpi
+ irqbalance.service
| \ 1210 irqbalance
+ avahi-daemon.service
| + 1175 avahi-daemon: running [epsilon.local]
+ NetworkManager.service
| + 1171 /usr/sbin/NetworkManager --no-daemon
| \ 4028 /sbin/dhclient -d -4 -sf /usr/libexec/nm-dhcp-client.action -pf /var/run/dhclient-wlan0.pid -lf /var/lib/dhclient/dhclient-7d32a784-ede9-4cf6-9ee3-6
+ rsyslog.service
| \ 1193 /sbin/rsyslogd -c 4
+ mdmonitor.service
| \ 1207 mdadm --monitor --scan -f --pid-file=/var/run/mdadm/mdadm.pid
+ cups.service
| \ 1195 cupsd -C /etc/cups/cupsd.conf
+ auditd.service
| + 1131 auditd
| + 1133 /sbin/audispd
| \ 1135 /usr/sbin/sedispatch
+ dbus.service
| +  1096 /bin/dbus-daemon --system --address=systemd: --nofork --systemd-activation
| +  1216 /usr/sbin/modem-manager
| +  1219 /usr/libexec/polkit-1/polkitd
| +  1242 /usr/sbin/wpa_supplicant -c /etc/wpa_supplicant/wpa_supplicant.conf -B -u -f /var/log/wpa_supplicant.log -P /var/run/wpa_supplicant.pid
| +  1453 /usr/libexec/upowerd
| +  1733 /usr/libexec/udisks-daemon
| +  1747 udisks-daemon: polling /dev/sr0
| \ 29509 /usr/libexec/packagekitd
+ dev-mqueue.mount
```

```
+ dev-hugepages.mount
\ sysinit.service
  +   455 /sbin/udevd -d
  +  4016 /usr/sbin/bluetoothd --udev
  + 28188 /sbin/udevd -d
  \ 28191 /sbin/udevd -d
```

(Как и предыдущий, этот листинг был сокращен за счет удаления перечня потоков ядра.)

Как видно из листинга, данная команда наглядно показывает принадлежность процессов к их контрольным группам, а следовательно, и к службам, так как systemd именует группы в соответствии с названиями служб. Например, из приведенного листинга нетрудно понять, что служба системного аудита `auditd.service` порождает три отдельных процесса: `auditd`, `audispd` и `sedispatch`.

Наиболее внимательные читатели, вероятно, уже заметили, что некоторые процессы помещены в группу `/user/lennart/1`. Дело в том, что systemd занимается отслеживанием и группировкой не только процессов, относящихся к системным службам, но и процессов, запущенных в рамках пользовательских сеансов. В последующих статьях мы обсудим этот вопрос более подробно.

# 3 Преобразование SysV init-скрипта в systemd service-файл

Традиционно, службы Unix и Linux (демоны) запускаются через SysV init-скрипты. Эти скрипты пишутся на языке Bourne Shell (`/bin/sh`), располагаются в специальном каталоге (обычно `/etc/rc.d/init.d/`) и вызываются с одним из стандартных параметров (`start`, `stop`, `reload` и т.п.) — таким образом указывается действие, которое необходимо произвести над службой (запустить, остановить, заставить перечитать конфигурацию). При запуске службы такой скрипт, как правило, вызывает бинарник демона, который, в свою очередь, форкается, порождая фоновый процесс (т.е. демонизируется). Заметим, что shell-скрипты, как правило, отличается низкой скоростью работы, излишней подробностью изложения и крайней хрупкостью. Читать их, из-за изобилия всевозможного вспомогательного и дополнительного кода, чрезвычайно тяжело. Впрочем, нельзя не упомянуть, что эти скрипты являются очень гибким инструментом (ведь, по сути, это всего лишь код, который можно модифицировать как угодно). С другой стороны, многие задачи, возникающие при работе со службами, довольно тяжело решить средствами shell-скриптов. К таким задачам относятся: организация параллельного исполнения, корректное отслеживание процессов, конфигурирование различных параметров среды исполнения процесса. systemd обеспечивает совместимость с init-скриптами, однако, с учетом описанных выше их недостатков, более правильным решением будет использование штатных service-файлов systemd для всех установленных в системе служб. Стоит отметить что, в отличие от init-скриптов, которые часто приходится модифицировать при переносе из одного дистрибутива в другой, один и тот же service-файл будет работать в любом дистрибутиве, использующем systemd (а таких дистрибутивов с каждым днем становится все больше и больше). Далее мы вкратце рассмотрим процесс преобразования SysV init-скрипта в service-файл systemd. Вообще говоря, service-файл должен создаваться разработчиками каждого демона, и включаться в комплект его поставки. Если вам удалось успешно создать работоспособный service-файл для какого-либо демона, настоятельно рекомендуем вам отправить этот файл разработчикам. Вопросы по полноценной интеграции демонов с systemd, с максимальным использованием всех его возможностей, будут рассмотрены в последующих статьях этого цикла, пока же ограничимся ссылкой на страницу официальной документации.

Итак, приступим. В качестве примера возьмем init-скрипт демона ABRT (Automatic Bug Reporting Tool, службы, занимающейся сбором crash dump'ов). Исходный скрипт (в варианте для дистрибутива Fedora) можно загрузить здесь.

Начнем с того, что прочитаем исходный скрипт (неожиданный ход, правда?) и выделим полезную информацию из груды хлама. Практически у всех init-скриптов бóльшая часть кода является чисто вспомогательной, и мало чем отличается от одного скрипта к другому. Как правило, при создании новых скриптов этот код просто копируется из уже существующих (разработка в стиле copy-paste). Итак, в исследуемом скрипте нас интересует следующая информация:

- Строка описания службы: «Daemon to detect crashing apps». Как нетрудно заметить, комментарии в заголовке скрипта весьма пространны и описывают не сколь-

ко саму службу, сколько скрипт, ее запускающий. service-файлы systemd тоже содержат описание, но оно относится исключительно к службе, а не к service-файлу.

- LSB-заголовок[6], содержащий информацию о зависимостях. systemd, базирующийся на идеях socket-активации, обычно не требует явного описания зависимостей (либо требует самого минимального описания). Заметим, что основополагающие принципы systemd, включая socket-активацию, рассмотрены в статье Rethinking PID 1, в которой systemd был впервые представлен широкой публике[7]. Возвращаясь к нашему примеру: в данном случае ценной информацией о зависимостях является только строка `Required-Start: $syslog`, сообщающая, что для работы abrtd требуется демон системного лога. Информация о второй зависимости, `$local_fs`, является избыточной, так как systemd приступает к запуску служб уже после того, как все файловые системы готовы для работы.

- Также, LSB-заголовок сообщает, что данная служба должна быть запущена на уровнях исполнения (runlevels) 3 (консольный многопользовательский) и 5 (графический многопользовательской).

- Исполняемый бинарник демона называется `/usr/sbin/abrtd`.

Вот и вся полезная информация. Все остальное содержимое 115-строчного скрипта является чисто вспомогательным кодом: операции синхронизации и упорядочивания запуска (код, относящийся к lock-файлам), вывод информационных сообщений (команды `echo`), разбор входных параметров (монструозный блок `case`).

На основе приведенной выше информации, мы можем написать следующий service-файл:

```
[Unit]
Description=Daemon to detect crashing apps
After=syslog.target

[Service]
ExecStart=/usr/sbin/abrtd
Type=forking

[Install]
WantedBy=multi-user.target
```

Рассмотрим этот файл поподробнее.

Секция `[Unit]` содержит самую общую информацию о службе. Не будем забывать, что systemd управляет не только службами, но и многими другими объектами, в частности, устройствами, точками монтирования, таймерами и т.п. Общее наименование всех этих объектов — юнит (unit). Одноименная секция конфигурационного файла определяет наиболее общие свойства, которые могут быть присущи любому юниту. В нашем случае это, во-первых, строка описания, и во-вторых, указание, что данный юнит рекомендуется активировать после запуска демона системного лога[8]. Эта информация,

[6] LSB-заголовок — определенная в Linux Standard Base схема записи метаданных о службах в блоках комментариев соответствующих init-скриптов. Изначально эта схема была введена именно для того, чтобы стандартизировать init-скрипты во всех дистрибутивах. Однако разработчики многих дистрибутивов не считают нужным точно исполнять требования LSB, и поэтому формы представления метаданных в различных дистрибутивах могут отличаться. Вследствие этого, при переносе init-скрипта из одного дистрибутива в другой, скрипт приходится модифицировать. Например, демон пересылки почты при описании зависимостей может именоваться `MTA` или `smtpdaemon` (Fedora), `smtp` (openSUSE), `mail-transport-agent` (Debian и Ubuntu), `mail-transfer-agent`. Таким образом, можно утверждать, что стандарт LSB не справляется с возложенной на него задачей.

[7] Прим. перев.: Ее русский перевод (сделанный совершенно независимо от данного документа, совсем другим человеком) можно прочитать здесь: часть 1, часть 2.

[8] Строго говоря, эту зависимость здесь указывать не нужно — в системах, в которых демон системного лога активируется через сокет, данная зависимость является избыточной. Современные реализации демона системного лога (например, rsyslog начиная с пятой версии) поддерживают активацию через сокет. В системах, использующих такие реализации, явное указание `After=syslog.target` будет из-

как мы помним, была указана в LSB-заголовке исходного init-скрипта. В нашем конфигурационном файле мы указываем зависимость от демона системного лога при помощи директивы `After`, указывающей на юнит `syslog.taget`. Это специальный юнит, позволяющий ссылаться на любую реализацию демона системного лога, независимо от используемой программы (например, rsyslog или syslog-ng) и типа активации (как обычной службы или через log-сокет). Подробнее о таких специальных юнитах можно почитать страницу официальной документации. Обратите внимание, что директива `After`, в отсутствие директивы `Requires`, задает лишь порядок загрузки, но не задает жесткой зависимости. То есть, если при загрузке конфигурация systemd будет предписывать запуск как демона системного лога, так и abrtd, то сначала будет запущен демон системного лога, и только потом abrtd. Если же конфигурация не будет содержать явного указания запустить демон системного лога, он не будет запущен даже при запуске abrtd. И это поведение нас полностью устраивает, так как abrtd прекрасно может обходиться и без демона системного лога. В противном случае, мы могли бы воспользоваться директивой `Requires`, задающей жесткую зависимость между юнитами.

Следующая секция, `[Service]`, содержит информацию о службе. Сюда включаются настройки, относящие именно к службам, но не к другим типам юнитов. В нашем случае, таких настроек две: `ExecStart`, определяющая расположение бинарника демона и аргументы, с которыми он будет вызван (в нашем случае они отсутствуют), и `Type`, позволяющая задать метод, по которому systemd определит окончание периода запуска службы. Традиционный для Unix метод демонизации процесса, когда исходный процесс форкается, порождая демона, после чего завершается, описывается типом `forking` (как в нашем случае). Таким образом, systemd считает службу запущенной с момента завершения работы исходного процесса, и рассматривает в качестве основного процесса этой службы порожденный им процесс-демон.

И наконец, третья секция, `[Install]`. Она содержит рекомендации по установке конкретного юнита, указывающие, в каких ситуациях он должен быть активирован. В нашем случае, служба abrtd запускается при активации юнита `multi-user.target`. Это специальный юнит, примерно соответствующий роли третьего уровня исполнения классического SysV[9]. Директива `WantedBy` никак не влияет на уже работающую службу, но она играет важную роль при выполнении команды `systemctl enable`, задавая, в каких условиях должен активироваться устанавливаемый юнит. В нашем примере, служба abrtd будет активироваться при переходе в состояние `multi-user.target`, т.е., при каждой нормальной[10] загрузке[11].

Вот и все. Мы получили минимальный рабочий service-файл systemd. Чтобы проверить его работоспособность, скопируем его в `/etc/systemd/system/abrtd.service`, после чего командой `systemctl daemon-reload` уведомим systemd об изменении конфигурации. Теперь нам остается только запустить нашу службу: `systemctl start abrtd.service`. Проверить состояние службы можно командой `systemctl status abrtd.service`, а чтобы остановить ее, нужно скомандовать `systemctl stop abrtd.service`. И наконец, команда `systemctl enable abrtd.service` выполнит установку service-файла, обеспечив его активацию при каждой загрузке (аналог `chkconfig abrtd on` в классическом SysV).

Приведенный выше service-файл является практический точным переводом исходного init-скрипта, и он никак не использует широкий спектр возможностей, предоставляемых systemd. Ниже приведен немного улучшенный вариант этого же файла:

---

быточным, так как соответствующая функциональность поддерживается автоматически. Однако, эту строчку все-таки стоит указать для обеспечения совместимости с системами, использующими устаревшие реализации демона системного лога.

[9] В том контексте, в котором он используется в большинстве дистрибутивов семейства Red Hat, а именно, многопользовательский режим без запуска графической оболочки.

[10] Прим. перев.: К «ненормальным» загрузкам можно отнести, например, загрузки в режимах `emergency.target` или `rescue.target` (является аналогом первого уровня исполнения в классической SysV).

[11] Обратите внимание, что режим графической загрузки в systemd (`graphical.target`, аналог runlevel 5 в SysV) является надстройкой над режимом многопользовательской консольной загрузки (`multi-user.target`, аналог runlevel 3 в SysV). Таким образом, все службы, запускаемые в режиме `multi-user.target`, будут также запускаться и в режиме `graphical.target`.

```
[Unit]
Description=ABRT Automated Bug Reporting Tool
After=syslog.target

[Service]
Type=dbus
BusName=com.redhat.abrt
ExecStart=/usr/sbin/abrtd -d -s

[Install]
WantedBy=multi-user.target
```

Чем же новый вариант отличается от предыдущего? Ну, прежде всего, мы уточнили описание службы. Однако, ключевым изменением является замена значения `Type` с `forking` на `dbus` и связанные с ней изменения: добавление имени службы в шине D-Bus (директива `BusName`) и задание дополнительных аргументов abrtd «`-d -s`». Но зачем вообще нужна эта замена? Каков ее практический смысл? Чтобы ответить на этот вопрос, мы снова возвращаемся к демонизации. В ходе данной операции процесс дважды форкается и отключается от всех терминалов. Это очень удобно при запуске демона через скрипт, но в случае использования таких продвинутых систем инициализации, как systemd, подобное поведение не дает никаких преимуществ, но вызывает неоправданные задержки. Даже если мы оставим в стороне вопрос скорости загрузки, останется такой важный аспект, как отслеживание состояния служб. systemd решает и эту задачу, контролируя работу службы и при необходимости реагируя на различные события. Например, при неожиданном падении основного процесса службы, systemd должен зарегистрировать идентификатор и код выхода процесса, также, в зависимости от настроек, он может попытаться перезапустить службу, либо активировать какой-либо заранее заданный юнит. Операция демонизации несколько затрудняет решение этих задач, так как обычно довольно сложно найти связь демонизированного процесса с исходным (собственно, смысл демонизации как раз и сводится к уничтожению этой связи) и, соответственно, для systemd сложнее определить, какой из порожденных в рамках данной службы процессов является основным. Чтобы упростить для него решение этой задачи, мы и воспользовались типом запуска `dbus`. Он подходит для всех служб, которые в конце процесса инициализации регистрируют свое имя на шине D-Bus[12]. ABRTd относится к ним. С новыми настройками, systemd запустит процесс abrtd, который уже не будет форкаться (согласно указанным нами ключам «`-d -s`»), и в качестве момента окончания периода запуска данной службы systemd будет рассматривать момент регистрации имени `com.redhat.abrt` на шине D-Bus. В этом случае основным для данной службы будет считаться процесс, непосредственно порожденный systemd. Таким образом, systemd располагает удобным методом для определения момента окончания запуска службы, а также может легко отслеживать ее состояние.

Собственно, это все, что нужно было сделать. Мы получили простой конфигурационный файл, в 10 строчках которого содержится больше полезной информации, чем в 115 строках исходного init-скрипта. Добавляя в наш файл по одной строчке, мы можем использовать различные полезные функции systemd, создание аналога которых в традиционном init-скрипте потребовало бы значительных усилий. Например, добавив строку `Restart=restart-always`, мы приказываем systemd автоматически перезапускать службу после каждого ее падения. Или, например, добавив `OOMScoreAdjust=-500`, мы попросим ядро сберечь эту службу, даже если OOM Killer выйдет на тропу войны. А если мы добавим строчку `CPUSchedulingPolicy=idle`, процесс abrtd будет работать только в те моменты, когда система больше ничем не занята, что позволит не создавать помех для процессов, активно использующих CPU.

За более подробным описанием всех опций настройки, вы можете обратиться к страницам руководства systemd.unit, systemd.service, systemd.exec. Полный список доступных страниц можно просмотреть здесь.

[12] В настоящее время практически все службы дистрибутива Fedora после запуска регистрируется на шине D-Bus.

Конечно, отнюдь не все init-скрипты так же легко преобразовать в service-файлы. Но, к счастью, «проблемных» скриптов не так уж и много.

# 4 Убить демона

Убить системного демона нетрудно, правда? Или... все не так просто?

Если ваш демон функционирует как один процесс, все действительно очень просто. Вы командуете `killall rsyslogd`, и демон системного лога останавливается. Впрочем, этот метод не вполне корректен, так как он действует не только на самого демона, но и на другие процессы с тем же именем. Иногда подобное поведение может привести к неприятным последствиям. Более правильным будет использование pid-файла: `kill $(cat /var/run/syslogd.pid)`. Вот, вроде бы, и все, что вам нужно... Или мы упускаем еще что-то?

Действительно, мы забываем одну простую вещь: существуют службы, такие, как Apache, crond, atd, которые по роду служебной деятельности должны запускать дочерние процессы. Это могут быть совершенно посторонние, указанные пользователем программы (например, задачи cron/at, CGI-скрипты) или полноценные серверные процессы (например, Apache workers). Когда вы убиваете основной процесс, он может остановить все дочерние процессы. А может и не остановить. В самом деле, если служба функционирует в штатном режиме, ее обычно останавливают командой `stop`. К прямому вызову `kill` администратор, как правило, прибегает только в аварийной ситуации, когда служба работает неправильно и может не среагировать на стандартную команду остановки. Таким образом, убив, например, основной сервер Apache, вы можете получить от него в наследство работающие CGI-скрипты, причем их родителем автоматически станет PID 1 (init), так что установить их происхождение будет не так-то просто.

systemd спешит к нам на помощь. Команда `systemctl kill` позволит отправить сигнал всем процессам, порожденным в рамках данной службы. Например:

```
# systemctl kill crond.service
```

Вы можете быть уверены, что всем процессам службы cron будет отправлен сигнал `SIGTERM`. Разумеется, можно отправить и любой другой сигнал. Скажем, если ваши дела совсем уж плохи, вы можете воспользоваться и `SIGKILL`:

```
# systemctl kill -s SIGKILL crond.service
```

После ввода этой команды, служба cron будет жестоко убита вместе со всеми ее дочерними процессами, вне зависимости от того, сколько раз она форкалась, и как бы она ни пыталась сбежать из-под нашего контроля при помощи двойного форка или форк-бомбардировки[13].

В некоторых случаях возникает необходимость отправить сигнал именно основному процессу службы. Например, используя `SIGHUP`, мы можем заставить демона перечитать файлы конфигурации. Разумеется, передавать HUP вспомогательным процессам в этом случае совершенно необязательно. Для решения подобной задачи неплохо подойдет и классический метод с pid-файлом, однако у systemd и на этот случай есть простое решение, избавляющее вас от необходимости искать нужный файл:

```
# systemctl kill -s HUP --kill-who=main crond.service
```

[13]Прим. перев.: Стоит особо отметить, что использование контрольных групп не только упрощает процесс уничтожения форк-бомб, но и значительно уменьшает ущерб от работающей форк-бомбы. Так как systemd автоматически помещает каждую службу и каждый пользовательский сеанс в свою контрольную группу по ресурсу процессорного времени, запуск форк-бомбы одним пользователем или службой не создаст значительных проблем с отзывчивостью системы у других пользователей и служб. Таким образом, в качестве основной угрозы форк-бомбардировки остаются лишь возможности исчерпания памяти и идентификаторов процессов (PID). Впрочем, и их тоже можно легко устранить: достаточно задать соответствующие лимиты в конфигурационном файле службы (см. systemd.exec(5)).

Итак, что же принципиально новое привносит systemd в рутинный процесс убийства демона? Прежде всего: впервые в истории Linux представлен способ принудительной остановки службы, не зависящий от того, насколько добросовестно основной процесс службы выполняет свои обязательства по остановке дочерних процессов. Как уже упоминалось выше, необходимость отправить процессу `SIGTERM` или `SIGKILL` обычно возникает именно в нештатной ситуации, когда вы уже не можете быть уверены, что демон корректно исполнит все свои обязанности.

После прочтения сказанного выше у вас может возникнуть вопрос: в чем разница между `systemctl kill` и `systemctl stop`? Отличие состоит в том, что `kill` просто отправляет сигнал заданному процессу, в то время как `stop` действует по «официально» определенному методу, вызывая команду, определенную в параметре `ExecStop` конфигурации службы. Обычно команды `stop` бывает вполне достаточно для остановки службы, и к `kill` приходится прибегать только в крайних случаях, например, когда служба «зависла» и не реагирует на команды.

Кстати говоря, при использовании параметра «`-s`», вы можете указывать названия сигналов как с префиксом SIG, так и без него — оба варианта будут работать.

В завершение стоит сказать, что для нас весьма интересным и неожиданным оказался тот факт, что до появления systemd в Linux просто не существовало инструментов, позволяющих корректно отправить сигнал службе в целом, а не отдельному процессу.

# 5 Три уровня выключения

В systemd существует три уровня (разновидности) действий, направленных на прекращение работы службы (или любого другого юнита):

- Вы можете *остановить* службу, то есть прекратить выполнение уже запущенных процессов службы. При этом сохраняется возможность ее последующего запуска, как ручного (через команду `systemctl start`), так и автоматического (при загрузке системы, при поступлении запроса через сокет или системную шину, при срабатывании таймера, при подключении соответствующего оборудования и т.д.). Таким образом, остановка службы является временной мерой, не дающей никаких гарантий на будущее.

  В качестве примера рассмотрим остановку службы NTPd (отвечающей за синхронизацию времени по сети):

  ```
  systemctl stop ntpd.service
  ```

  Аналогом этой команды в классическом SysV init является

  ```
  service ntpd stop
  ```

  Заметим, что в Fedora 15, использующей в качестве системы инициализации systemd, в целях обеспечения обратной совместимости допускается использование классических SysV-команд, и systemd будет корректно воспринимать их. В частности, вторая приведенная здесь команда будет эквивалентна первой.

- Вы можете *отключить* службу, то есть отсоединить ее от всех триггеров активации. В результате служба уже не будет автоматически запускаться ни при загрузке системы, ни при обращении к сокету или адресу на шине, ни при подключении оборудования, и т.д. Но при этом сохраняется возможность «ручного» запуска службы (командой `systemctl start`). Обратите внимание, что при отключении уже запущенной службы, ее выполнение в текущем сеансе не останавливается — это нужно сделать отдельно, иначе процессы службы будут работать до момента выключения системы (но при следующем включении, разумеется, уже не запустятся).

  Рассмотрим отключение службы на примере все того же NTPd:

  ```
  systemctl disable ntpd.service
  ```

В классических SysV-системах аналогичная команда будет иметь вид

```
chkconfig ntpd off
```

Как и в предыдущем случае, в Fedora 15 вторая из этих команд будет действовать аналогично первой.

Довольно часто приходится сочетать действия отключения и остановки службы — такая комбинированная операция гарантирует, что уже исполняющиеся процессы службы будут прекращены, и служба больше не будет запускаться автоматически (но может быть запущена вручную):

```
systemctl disable ntpd.service
systemctl stop ntpd.service
```

Подобное сочетание команд используется, например, при деинсталляции пакетов в Fedora.

Обратите внимание, что отключение службы является перманентной мерой, и действует вплоть до явной отмены соответствующей командой. Перезагрузка системы не отменяет отключения службы.

- Вы можете *заблокировать* (замаскировать) службу. Действие этой операции аналогично отключению, но дает более сильный эффект. Если при отключении отменяется только возможность автоматического запуска службы, но сохраняется возможность ручного запуска, то при блокировке исключаются обе эти возможности. Отметим, что использование данной опции при непонимании принципов ее работы может привести к трудно диагностируемым ошибкам.

  Тем не менее, рассмотрим пример блокировки все той же службы NTPd:

  ```
  ln -s /dev/null /etc/systemd/system/ntpd.service
  systemctl daemon-reload
  ```

  Итак, блокировка сводится к созданию символьной ссылки с именем соответствующей службы, указывающей на `/dev/null`[14]. После такой операции служба не может быть запущена ни вручную, ни автоматически. Символьная ссылка создается в каталоге `/etc/systemd/system/`, а ее имя должно соответствовать имени файла описания службы из каталога `/lib/systemd/system/` (в нашем случае `ntpd.service`).

  Заметим, что systemd читает файлы конфигурации из обоих этих каталогов, но файлы из `/etc` (управляемые системным администратором) имеют приоритет над файлами из `/lib` (которые управляются пакетным менеджером). Таким образом, создание символьной ссылки (или обычного файла) `/etc/systemd/system/ntpd.service` предотвращает чтение штатного файла конфигурации `/lib/systemd/system/ntpd.service`.

  В выводе `systemctl status` заблокированные службы отмечаются словом `masked`. Попытка запустить такие службы командой `systemctl start` завершится ошибкой.

  В рамках классического SysV init, штатная реализация такой возможности отсутствует. Похожий эффект может быть достигнут с помощью «костылей», например, путем добавления команды `exit 0` в начало init-скрипта. Однако, подобные решения имеют ряд недостатков, например, потенциальная возможность конфликтов с пакетным менеджером (при очередном обновлении исправленный скрипт может быть просто затерт соответствующим файлом из пакета).

[14] Прим. перев.: Впоследствии в программу `systemctl` была добавлена поддержка команд `mask` и `unmask`, упрощающих процесс блокирования и разблокирования юнитов. Например, для блокирования службы `ntpd.service` теперь достаточно команды `systemctl mask ntpd.service`. Фактически она сделает то же самое, что и приведенная выше команда `ln`.

Стоит отметить, что блокировка службы, как и ее отключение, является перманентной мерой[15].

После прочтения изложенного выше, у читателя может возникнуть вопрос: как отменить произведенные изменения? Что ж, ничего сложного тут нет: `systemctl start` отменяет действия `systemctl stop`, `systemctl enable` отменяет действие `systemctl disable`, а `rm` отменяет действие `ln`.

# 6 Смена корня

Практически все администраторы и разработчики рано или поздно встречаются с chroot-окружениями. Системный вызов `chroot()` позволяет задать для определенного процесса (и его потомков) каталог, который они будут рассматривать как корневой `/`, тем самым ограничивая для них область видимости иерархии файловой системы отдельной ветвью. Большинство применений chroot-окружений можно отнести к двум классам задач:

1. Обеспечение безопасности. Потенциально уязвимый демон chroot'ится в отдельный каталог и, даже в случае успешной атаки, взломщик увидит лишь содержимое этого каталога, а не всю файловую систему — он окажется в ловушке chroot'а.

2. Подготовка и управление образом операционной системы при отладке, тестировании, компиляции, установке или восстановлении. При этом вся иерархия файловых систем гостевой ОС монтируется или создается в каталоге системы-хоста, и при запуске оболочки (или любого другого приложения) внутри этой иерархии, в качестве корня используется данный каталог. Система, которую «видят» такие программы, может сильно отличаться от ОС хоста. Например, это может быть другой дистрибутив, или даже другая аппаратная архитектура (запуск i386-гостя на x86_64-хосте). Гостевая ОС не может увидеть полного дерева каталогов ОС хоста.

В системах, использующих классический SysV init, использовать chroot-окружения сравнительно несложно. Например, чтобы запустить выбранного демона внутри дерева каталогов гостевой ОС, достаточно смонтировать внутри этого дерева `/proc`, `/sys` и остальные API ФС, воспользоваться программой `chroot(1)` для входа в окружение, и выполнить соответствующий init-скрипт, запустив `/sbin/service` внутри окружения.

Но в системах, использующих systemd, уже не все так просто. Одно из важнейших достоинств systemd состоит в том, что параметры среды, в которой запускаются демоны, никак не зависят от метода их запуска. В системах, использующих SysV init, многие параметры среды выполнения (в частности, лимиты на системные ресурсы, переменные окружения, и т.п.) наследуются от оболочки, из которой был запущен init-скрипт. При использовании systemd ситуация меняется радикально: пользователь просто уведомляет процесс init о необходимости запустить ту или иную службу, и тот запускает демона в чистом, созданном «с нуля» и тщательно настроенном окружении, параметры которого никак не зависят от настроек среды, из которой была отдана команда. Такой подход полностью отменяет традиционный метод запуска демонов в chroot-окружениях: теперь демон порождается процессом init (PID 1) и наследует корневой каталог от него, вне зависимости от того, находился ли пользователь, отдавший команду на запуск, в chroot-окружении, или нет. Кроме того, стоит особо отметить, что взаимодействие управляющих программ с systemd происходит через сокеты, находящиеся в каталоге `/run/systemd`, так что программы, запущенные в chroot-окружении, просто не смогут

[15] Прим. перев.: Подробно описав принцип работы блокировки службы (юнита), автор забывает привести практические примеры ситуаций, когда эта возможность оказывается полезной.

В частности, иногда бывает необходимо полностью предотвратить запуск службы в любой ситуации. При этом не стоит забывать, что в post-install скриптах пакетного менеджера или, скажем, в заданиях cron, вместо `systemctl try-restart` (`service condrestart`) может быть ошибочно указано `systemctl restart` (`service restart`), что является прямым указанием на запуск службы, если она еще не запущена. Вследствие таких ошибок, отключенная служба может «ожить» в самый неподходящий момент.

взаимодействовать с init-подсистемой (и это, в общем, неплохо, а если такое ограничение будет создавать проблемы, его можно легко обойти, используя bind-монтирование).

В свете вышесказанного, возникает вопрос: как правильно использовать chroot-окружения в системах на основе systemd? Что ж, постараемся дать подробный и всесторонний ответ на этот вопрос.

Для начала, рассмотрим первое из перечисленных выше применений chroot: изоляция в целях безопасности. Прежде всего, стоит заметить, что защита, предоставляемая chroot'ом, весьма эфемерна и ненадежна, так как chroot не является «дорогой с односторонним движением». Выйти из chroot-окружения сравнительно несложно, и соответствующее предупреждение даже присутствует на странице руководства. Действительно эффективной защиты можно достичь, только сочетая chroot с другими методиками. В большинстве случаев, это возможно только при наличии поддержки chroot в самой программе. Прежде всего, корректное конфигурирование chroot-окружения требует глубокого понимания принципов работы программы. Например, нужно точно знать, какие каталоги нужно bind-монтировать из основной системы, чтобы обеспечить все необходимые для работы программы каналы связи. С учетом вышесказанного, эффективная chroot-защита обеспечивается только в том случае, когда она реализована в коде самого демона. Именно разработчик лучше других знает (*обязан* знать), как правильно сконфигурировать chroot-окружение, и какой минимальный набор файлов, каталогов и файловых систем необходим внутри него для нормальной работы демона. Уже сейчас существуют демоны, имеющие встроенную поддержку chroot. К сожалению, в системе Fedora, установленной с параметрами по умолчанию, таких демонов всего два: Avahi и RealtimeKit. Оба они написаны одним очень хитрым человеком ;-) (Вы можете собственноручно убедиться в этом, выполнив команду `ls -l /proc/*/root`.)

Возвращаясь к теме нашего обсуждения: разумеется, systemd позволяет помещать выбранных демонов в chroot, и управлять ими точно так же, как и остальными. Достаточно лишь указать параметр `RootDirectory=` в соответствующем service-файле. Например:

```
[Unit]
Description=A chroot()ed Service

[Service]
RootDirectory=/srv/chroot/foobar
ExecStartPre=/usr/local/bin/setup-foobar-chroot.sh
ExecStart=/usr/bin/foobard
RootDirectoryStartOnly=yes
```

Рассмотрим этот пример подробнее. Параметр `RootDirectory=` задает каталог, в который производится chroot перед запуском исполняемого файла, заданного параметром `ExecStart=`. Заметим, что путь к этому файлу должен быть указан относительно каталога chroot (так что, в нашем случае, с точки зрения основной системы, на исполнение будет запущен файл `/srv/chroot/foobar/usr/bin/foobard`). Перед запуском демона будет вызван сценарий оболочки `setup-foobar-chroot.sh`, который должен обеспечить подготовку chroot-окружения к запуску демона (например, смонтировать в нем `/proc` и/или другие файловые системы, необходимые для работы демона). Указав `RootDirectoryStartOnly=yes`, мы задаем, что `chroot()` будет выполняться только перед выполнением файла из `ExecStart=`, а команды из других директив, в частности, `ExecStartPre=`, будут иметь полный доступ к иерархии файловых систем ОС (иначе наш скрипт просто не сможет выполнить bind-монтирование нужных каталогов). Более подробную информацию по опциям конфигурации вы можете получить на страницах руководства.

Поместив приведенный выше текст примера в файл `/etc/systemd/system/foobar.service`, вы сможете запустить chroot'нутого демона командой `systemctl start foobar.service`. Информацию о его текущем состоянии можно получить с помощью команды `systemctl status foobar.service`. Команды управления и мониторинга службы не зависят от того, запущена ли она в chroot'е, или нет. Этим systemd отличается от классического SysV init.

Новые ядра Linux поддерживают возможность создания независимых пространств имен файловых систем (в дальнейшем FSNS, от «file system namespaces»). По функциональности этот механизм аналогичен `chroot()`, однако предоставляет гораздо более широкие возможности, и в нем отсутствуют проблемы с безопасностью, характерные для chroot. systemd позволяет использовать при конфигурировании юнитов некоторые возможности, предоставляемые FSNS. В частности, использование FSNS часто является гораздо более простой и удобной альтернативой созданию полновесных chroot-окружений. Используя директивы `ReadOnlyDirectories=`, `InaccessibleDirectories=`, вы можете задать ограничения по использованию иерархии файловых систем для заданной службы: ее корнем будет системный корневой каталог, однако указанные в этих директивах подкаталоги будут доступны только для чтения или вообще недоступны для нее. Например:

```
[Unit]
Description=A Service With No Access to /home

[Service]
ExecStart=/usr/bin/foobard
InaccessibleDirectories=/home
```

Такая служба будет иметь доступ ко всей иерархии файловых систем ОС, с единственным исключением — она не будет видеть каталог `/home`, что позволит защитить данные пользователей от потенциальных хакеров. (Подробнее об этих опциях можно почитать на странице руководства.)

Фактически, FSNS по множеству параметров превосходят `chroot()`. Скорее всего, Avahi и RealtimeKit в ближайшем будущем перейдут от `chroot()` к использованию FSNS.

Итак, мы рассмотрели вопросы использования chroot для обеспечения безопасности. Переходим ко второму пункту: подготовка и управление образом операционной системы при отладке, тестировании, компиляции, установке или восстановлении.

chroot-окружения, по сути, весьма примитивны: они изолируют только иерархии файловых систем. Даже после chroot'а в определенный подкаталог, процесс попрежнему имеет полный доступ к системным вызовам, может убивать процессы, запущенные в основной системе, и т.п. Вследствие этого, запуск полноценной ОС (или ее части) внутри chroot'а несет угрозу для хост-системы: у гостя и хоста отличается лишь содержимое файловой системы, все остальное у них общее. Например, если вы обновляете дистрибутив, установленный в chroot-окружении, и пост-установочный скрипт пакета отправляет `SIGTERM` процессу init для его перезапуска[16], на него среагирует именно хост-система! Кроме того, хост и chroot'нутая система будут иметь общую разделяемую память SysV (SysV shared memory), общие сокеты из абстрактных пространств имен (abstract namespace sockets) и другие элементы IPC. Для отладки, тестирования, компиляции, установки и восстановлении ОС не требуется абсолютно неуязвимая изоляция — нужна лишь надежная защита от *случайного* воздействия на ОС хоста изнутри chroot-окружения, иначе вы можете получить целый букет проблем, как минимум, от пост-инсталляционных скриптов при установке пакетов в chroot-окружении.

systemd имеет целый ряд возможностей, полезных для работы с chroot-системами:

Прежде всего, управляющая программа `systemctl` автоматически определяет, что она запущена в chroot-системе. В такой ситуации будут работать только команды `systemctl enable` и `systemctl disable`, во всех остальных случаях `systemctl` просто не будет ничего делать, возвращая код успешного завершения операции. Таким образом, пакетные скрипты смогут включить/отключить запуск «своих» служб при загрузке (или в других ситуациях), однако команды наподобие `systemctl restart` (обычно выполняется при обновлении пакета) не дадут никакого эффекта внутри chroot-окружения[17].

[16] Прим. перев.: Во избежание путаницы отметим, что перезапуск процесса init (PID 1) «на лету» при получении `SIGTERM` поддерживается только в systemd, в классическом SysV init такой возможности нет.

[17] Прим. перев.: Автор забывает отметить не вполне очевидный момент: такое поведение `systemctl` проявляется только в «мертвых» окружениях, т.е. в тех, где не запущен процесс init, и соответственно

Однако, куда более интересные возможности предоставляет программа systemd-nspawn, входящая в стандартный комплект поставки systemd. По сути, это улучшенный аналог `chroot(1)` — она не только подменяет корневой каталог, но и создает отдельные пространства имен для дерева файловых систем (FSNS) и для идентификаторов процессов (PID NS), предоставляя легковесную реализацию системного контейнера[18]. `systemd-nspawn` проста в использовании как `chroot(1)`, однако изоляция от хост-системы является более полной и безопасной. Всего одной командой вы можете загрузить внутри контейнера *полноценную* ОС (на базе systemd или SysV init). Благодаря использованию независимых пространств идентификаторов процессов, процесс init внутри контейнера получит PID 1, что позволит работать ему в штатном режиме. Также, в отличие от `chroot(1)`, внутри окружения будут автоматически смонтированы `/proc` и `/sys`.

Следующий пример иллюстрирует возможность запустить Debian на Fedora-хосте всего тремя командами:

```
# yum install debootstrap
# debootstrap --arch=amd64 unstable debian-tree/
# systemd-nspawn -D debian-tree/
```

Вторая из этих команд обеспечивает развертывание в подкаталоге `./debian-tree/` файловой структуры дистрибутива Debian, после чего третья команда запускает внутри полученной системы процесс командной оболочки. Если вы хотите запустить внутри контейнера полноценную ОС, воспользуйтесь командой

```
# systemd-nspawn -D debian-tree/ /sbin/init
```

После быстрой загрузки вы получите приглашение оболочки, запущенной внутри полноценной ОС, функционирующей в контейнере. Изнутри контейнера невозможно увидеть процессы, которые находятся вне его. Контейнер сможет пользоваться сетью хоста[19], однако не имеет возможности изменить ее настройки (это может привести к серии ошибок в процессе загрузки гостевой ОС, но ни одна из этих ошибок не должна быть критической). Контейнер получает доступ к `/sys` и `/proc/sys`, однако, во избежание вмешательства контейнера в конфигурацию ядра и аппаратного обеспечения хоста, эти каталоги будут смонтированы только для чтения. Обратите внимание, что эта защита блокирует лишь *случайные*, *непредвиденные* попытки изменения параметров. При необходимости, процесс внутри контейнера, обладающий достаточными полномочиями, сможет перемонтировать эти файловые системы в режиме чтения-записи.

Итак, что же такого хорошего в `systemd-nspawn`?

1. Использовать эту утилиту очень просто. Вам даже не нужно вручную монтировать внутри окружения `/proc` и `/sys` — она сделает это за вас, а ядро автоматически отмонтирует их, когда последний процесс контейнера завершится.

2. Обеспечивается надежная изоляция, предотвращающая случайные изменения параметров ОС хоста изнутри контейнера.

3. Теперь вы можете загрузить внутри контейнера полноценную ОС, а не одну-единственную оболочку.

---

отсутствуют управляющие сокеты в `/run/systemd`. Такая ситуация возникает, например, при установке системы в chroot через debootstrap/febootstrap. В этом случае возможности `systemctl` ограничиваются операциями с символьными ссылками, определяющими триггеры активации юнитов, т.е. выполнением действий `enable` и `disable`, не требующих непосредственного взаимодействия с процессом init.

[18] Прим. перев.: Используемые в `systemd-nspawn` механизмы ядра Linux, такие, как FS NS и PID NS, также лежат в основе LXC, системы контейнерной изоляции для Linux, которая позиционируется как современная альтернатива классическому OpenVZ. Стоит отметить, что LXC ориентирована прежде всего на создание независимых виртуальных окружений, с поддержкой раздельных сетевых стеков, ограничением на ресурсы, сохранением настроек и т.п., в то время как `systemd-nspawn` является лишь более удобной и эффективной заменой команды `chroot(1)`, предназначенной прежде всего для развертывания, восстановления, сборки и тестирования операционных систем. Далее автор разъясняет свою точку зрения на этот вопрос.

[19] Прим. перев.: Впоследствии в `systemd-nspawn` была добавлена опция `--private-network`, позволяющая изолировать систему внутри контейнера от сети хоста: такая система будет видеть только свой собственный интерфейс обратной петли.

4. Эта утилита очень компактна и присутствует везде, где установлен systemd. Она не требует специальной установки и настройки.

systemd уже подготовлен для работы внутри таких контейнеров. Например, когда подается команда на выключение системы внутри контейнера, systemd на последнем шаге вызывает не `reboot()`, а просто `exit()`.

Стоит отметить, что `systemd-nspawn` все же не является полноценной системой контейнерной виртуализации/изоляции — если вам нужно такое решение, воспользуйтесь LXC. Этот проект использует те же самые механизмы ядра, но предоставляет куда более широкие возможности, включая виртуализацию сети. Могу предложить такую аналогию: `systemd-nspawn` как реализация контейнера похожа на GNOME 3 — компактна и проста в использовании, опций для настройки очень мало. В то время как LXC больше похож на KDE: опций для настройки больше, чем строк кода. Я создал `systemd-nspawn` специально для тестирования, отладки, сборки, восстановления. Именно для этих задач вам стоит ее использовать — она неплохо с ними справляется, куда лучше, чем `chroot(1)`.

Что ж, пора заканчивать. Итак:

1. Использование `chroot()` для обеспечения безопасности дает наилучший результат, когда оно реализовано непосредственно в коде самой программы.

2. `ReadOnlyDirectories=` и `InaccessibleDirectories=` могут быть удобной альтернативой созданию полновесных chroot-окружений.

3. Если вам нужно поместить в chroot-окружение какую-либо службу, воспользуйтесь опцией `RootDirectory=`.

4. `systemd-nspawn` — очень неплохая штука.

5. chroot'ы убоги, FSNS — 1337.

И все это уже сейчас доступно в Fedora 15.

# 7 Поиск виновных

Fedora 15[20] является первым релизом Fedora, использующим systemd в качестве системы инициализации по умолчанию. Основной нашей целью при работе над выпуском F15 является обеспечение полной взаимной интеграции и корректной работы всех компонентов. При подготовке следующего релиза, F16, мы сконцентрируемся на дальнейшей полировке и ускорении системы. Для этого мы подготовили ряд инструментов (доступных уже в F15), которые должны помочь нам в поиске проблем, связанных с процессом загрузки. В этой статье я попытаюсь рассказать о том, как найти виновников медленной загрузки вашей системы, и о том, что с ними делать дальше.

Первый инструмент, который мы можем вам предложить, очень прост: по завершении загрузки, systemd регистрирует в системном журнале информацию о суммарном времени загрузки:

```
systemd[1]: Startup finished in 2s 65ms 924us (kernel) + 2s 828ms 195us (initrd)
+ 11s 900ms 471us (userspace) = 16s 794ms 590us.
```

Эта запись означает следующее: на инициализацию ядра (до момента запуска initrd, т.е. dracut) ушло 2 секунды. Далее, чуть менее трех секунд работал initrd. И наконец, почти 12 секунд было потрачено systemd на запуск программ из пространства пользователя. Итоговое время, начиная с того момента, как загрузчик передал управление коду ядра, до того момента, как systemd завершил все операции, связанные с загрузкой системы, составило почти 17 секунд. Казалось бы, смысл этого числа вполне очевиден... Однако не стоит делать поспешных выводов. Прежде всего, сюда не входит время, затраченное на инициализацию вашего сеанса в GNOME, так как эта задача уже выходит

[20] Величайший в истории релиз свободной ОС.

за рамки задач процесса init. Кроме того, в этом показателе учитывается только время работы systemd, хотя часто бывает так, что некоторые демоны продолжают *свою* работу по инициализации уже после того, как секундомер остановлен. Проще говоря: приведенные числа позволяют лишь оценить общую скорость загрузки, однако они не являются точной характеристикой длительности процесса.

Кроме того, эта информация носит поверхностный характер: она не сообщает, какие именно системные компоненты заставляют systemd ждать так долго. Чтобы исправить это упущение, мы ввели команду `systemd-analyze blame`:

```
$ systemd-analyze blame
  6207ms udev-settle.service
  5228ms cryptsetup@luks\x2d9899b85d\x2df790\x2d4d2a\x2da650\x2d8b7d2fb92cc3.service
   735ms NetworkManager.service
   642ms avahi-daemon.service
   600ms abrtd.service
   517ms rtkit-daemon.service
   478ms fedora-storage-init.service
   396ms dbus.service
   390ms rpcidmapd.service
   346ms systemd-tmpfiles-setup.service
   322ms fedora-sysinit-unhack.service
   316ms cups.service
   310ms console-kit-log-system-start.service
   309ms libvirtd.service
   303ms rpcbind.service
   298ms ksmtuned.service
   288ms lvm2-monitor.service
   281ms rpcgssd.service
   277ms sshd.service
   276ms livesys.service
   267ms iscsid.service
   236ms mdmonitor.service
   234ms nfslock.service
   223ms ksm.service
   218ms mcelog.service
...
```

Она выводит список юнитов systemd, активированных при загрузке, с указанием времени инициализации для каждого из них. Список отсортирован по убыванию этого времени, поэтому наибольший интерес для нас представляют первые строчки. В нашем случае это `udev-settle.service` и `cryptsetup@luks\x2d9899b85d\x2df790\x2d4d2a\x2da650\x2d8b7d2fb92cc3.service`, инициализация которых занимает более одной секунды. Стоит отметить, что к анализу вывода команды `systemd-analyze blame` тоже следует подходить с осторожностью: она не поясняет, *почему* тот или иной юнит тратит столько-то времени, она лишь констатирует факт, что время было затрачено. Кроме того, не стоит забывать, что юниты могут запускаться параллельно. В частности, если две службы были запущены одновременно, то время их инициализации будет значительно меньше, чем сумма времен инициализации каждой из них.

Рассмотрим повнимательнее первого осквернителя нашей загрузки: службу `udev-settle.service`[21] Почему ей требуется так много времени для запуска, и что мы можем с этим сделать? Данная служба выполняет очень простую функцию: она ожидает, пока udev завершит опрос устройств, после чего завершается. Опрос же устройств может занимать довольно много времени. Например, в нашем случае опрос устройств длится более 6 секунд из-за подключенного к компьютеру 3G-модема, в котором отсутствует SIM-карта. Этот модем очень долго отвечает на запросы udev. Опрос

[21] Прим. перев.: После объединения проектов systemd и udev, эта служба называется `systemd-udev-settle.service`.

устройств является частью схемы, обеспечивающей работу ModemManager'а и позволяющей NetworkManager'у упростить для вас настройку 3G. Казалось бы, очевидно, что виновником задержки является именно ModemManager, так как опрос устройств для него занимает слишком много времени. Но такое обвинение будет заведомо ошибочным. Дело в том, что опрос устройств очень часто оказывается довольно длительной процедурой. Медленный опрос 3G-устройств для ModemManager является частным случаем, отражающим это общее правило. Хорошая система опроса устройств обязательно должна учитывать тот факт, что операция опроса любого из устройств может затянуться надолго. Истинной причиной наших проблем является необходимость ожидать завершения опроса, т.е., наличие службы `udev-settle.service` как обязательной части нашего процесса загрузки.

Но почему эта служба вообще присутствует в нашем процессе загрузки? На самом деле, мы можем прекрасно обойтись и без нее. Она нужна лишь как часть используемой в Fedora схемы инициализации устройств хранения, а именно, набора скриптов, выполняющих настройку LVM, RAID и multipath-устройств. На сегодняшний день, реализация этих систем не поддерживает собственного механизма поиска и опроса устройств, и поэтому их запуск должен производиться только после того, как эта работа уже проделана — тогда они могут просто последовательно перебирать все диски. Однако, такой подход противоречит одному из фундаментальных требований к современным компьютерам: возможности подключать и отключать оборудование в любой момент, как при выключенном компьютере, так и при включенном. Для некоторых технологий невозможно точно определить момент завершения формирования списка устройств (например, это характерно для USB и iSCSI), и поэтому процесс опроса таких устройств обязательно должен включать некоторую фиксированную задержку, гарантирующую, что все подключенные устройства успеют отозваться. С точки зрения скорости загрузки это, безусловно, негативный эффект: соответствующие скрипты заставляют нас ожидать завершения опроса устройств, хотя большинство этих устройств не являются необходимыми на данной стадии загрузки. В частности, в рассматриваемой нами системе LVM, RAID и multipath вообще не используются![22]

С учетом вышесказанного, мы можем спокойно исключить `udev-settle.service` из нашего процесса загрузки, так как мы не используем ни LVM, ни RAID, ни multipath. Замаскируем эти службы, чтобы увеличить скорость загрузки:

```
# ln -s /dev/null /etc/systemd/system/udev-settle.service
# ln -s /dev/null /etc/systemd/system/fedora-wait-storage.service
# ln -s /dev/null /etc/systemd/system/fedora-storage-init.service
# systemctl daemon-reload
```

После перезагрузки мы видим, что загрузка стала на одну секунду быстрее. Но почему выигрыш оказался таким маленьким? Из-за второго осквернителя нашей загрузки — cryptsetup. На рассматриваемой нами системе зашифрован раздел `/home`. Специально для нашего тестирования я записал пароль в файл, чтобы исключить влияние скорости ручного набора пароля. К сожалению, для подключения шифрованного раздела cryptsetup требует более пяти секунд. Будем ленивы, и вместо того, чтобы исправлять

[22] Наиболее правильным решением в данном случае будет ожидание событий подключения устройств (реализованное через libudev или аналогичную технологию) с соответствующей обработкой каждого такого события — подобный подход позволит нам продолжить загрузку, как только будут готовы все устройства, необходимые для ее продолжения. Для того, чтобы загрузка была быстрой, мы должны ожидать завершения инициализации только тех устройств, которые *действительно* необходимы для ее продолжения на данной стадии. Ожидать остальные устройства в данном случае смысла нет. Кроме того, стоит отметить, что в числе программ, не приспособленных для работы с динамически меняющимся оборудованием и построенных в предположении о неизменности списка устройств, кроме служб хранения, есть и другие подсистемы. Например, в нашем случае работа initrd занимает так много времени главным образом из-за того, что для запуска Plymouth необходимо дождаться завершения инициализации всех устройств видеовывода. По неизвестной причине (во всяком случае, неизвестной для меня) подгрузка модулей для моих видеокарт Intel занимает довольно продолжительное время, что приводит к беспричинной задержке процесса загрузки. (Нет, я возражаю не против опроса устройств как такового, но против необходимости ожидать его завершения, чтобы продолжить загрузку.)

ошибку в cryptsetup[23], попробуем найти обходной путь[24]. Во время загрузки systemd должен дождаться, пока все файловые системы, перечисленные в `/etc/fstab` (кроме помеченных как `noauto`) будут обнаружены, проверены и смонтированы. Только после этого systemd может продолжить загрузку и приступить к запуску служб. Но первое обращение к `/home` (в отличие, например, от `/var`), происходит на поздней стадии процесса загрузки (когда пользователь входит в систему). Следовательно, нам нужно сделать так, чтобы этот каталог автоматически монтировался при загрузке, но процесс загрузки не ожидал завершения работы `cryptsetup`, `fsck` и `mount` для этого раздела. Как же сделать точку монтирования доступной, не ожидая, пока завершится процесс монтирования? Этого можно достичь, воспользовавшись магической силой systemd — просто добавим опцию монтирования `comment=systemd.automount` в `/etc/fstab`. После этого, systemd будет создавать в `/home` точку автоматического монтирования, и при первом же обращении к этому каталогу, если файловая система еще не будет готова к работе, systemd подготовит соответствующее устройство, проверит и смонтирует ее.

После внесения изменений в `/etc/fstab` и перезагрузки мы получаем:

```
systemd[1]: Startup finished in 2s 47ms 112us (kernel) + 2s 663ms 942us (initrd)
+ 5s 540ms 522us (userspace) = 10s 251ms 576us.
```

Прекрасно! Несколькими простыми действиями мы выиграли почти семь секунд. И эти два изменения исправляют только две наиболее очевидные проблемы. При более аккуратном и детальном исследовании, обнаружится еще множество моментов, которые можно улучшить. Например, на другом моем компьютере, лаптопе X300 двухлетней давности (и даже два года назад он был не самым быстрым на Земле), после небольшой доработки, время загрузки до полноценной среды GNOME составило около четырех секунд, и это еще не предел совершенства.

`systemd-analyze blame` — простой и удобный инструмент для поиска медленно запускающихся служб, однако он обладает одним существенным недостатком: он не показывает, насколько эффективно параллельный запуск снижает потерю времени для медленно запускающихся служб. Чтобы вы могли наглядно оценить этот фактор, мы подготовили команду `systemd-analyze plot`. Использовать ее очень просто:

```
$ systemd-analyze plot > plot.svg
$ eog plot.svg
```

Она создает наглядные диаграммы, показывающие моменты запуска служб и время, затраченное на их запуск, по отношению к другим службам. На текущий момент, она не показывает явно, кто кого ожидает, но догадаться обычно несложно.

Чтобы продемонстрировать эффект, порожденный двумя нашими оптимизациями, приведем ссылки на соответствующие графики: до и после (для полноты описания приведем также и соответствующие данные `systemd-analyze blame`: до и после).

У наиболее эрудированных читателей может возникнуть вопрос: как это соотносится с программой bootchart? Действительно, `systemd-analyze plot` и `bootchart` рисуют похожие графики. Однако, bootchart является намного более мощным инструментом — он детально показывает, что именно происходило во время загрузки, и отображает соответствующие графики использования процессора и ввода-вывода. `systemd-analyze plot` оперирует более высокоуровневой информацией: сколько времени затратила та или иная служба во время запуска, и какие службы были вынуждены ее ожидать. Используя оба этих инструмента, вы значительно упростите себе поиск причин замедления вашей загрузки.

Но прежде, чем вы вооружитесь описанными здесь средствами и начнете отправлять багрепорты авторам и сопровождающим программ, которые замедляют вашу загрузку,

[23] На самом деле, я действительно пытался исправить эту ошибку. Задержки в cryptsetup, по моим наблюдениям, обусловлены, главным образом, слишком большим значением по умолчанию для опции `--iter-time`. Я попытался доказать сопровождающим пакета cryptsetup, что снижение этого времени с 1 секунды до 100 миллисекунд не приведет к трагическим последствиям для безопасности, однако они мне не поверили.

[24] Вообще-то я предпочитаю решать проблемы, а не искать пути для их обхода, однако в данном конкретном случае возникает отличная возможность продемонстрировать одну интересную возможность systemd...

обдумайте все еще раз. Эти инструменты предоставляют вам «сырую» информацию. Постарайтесь не ошибиться, интерпретируя ее. Например, в при рассмотрении приведенного выше примера я показал, что проблема была вовсе не в `udev-settle.service`, и не в опросе устройств для ModemManager'а, а в неудачной реализации подсистемы хранения, требующей ожидать окончания опроса устройств для продолжения загрузки. Именно эту проблему и нужно исправлять. Поэтому постарайтесь правильно определить источник проблем. Возлагайте вину на тех, кто действительно виноват.

Как уже говорилось выше, все три описанных здесь инструмента доступны в Fedora 15 «из коробки».

Итак, какие же выводы мы можем сделать из этой истории?

- `systemd-analyze` — отличный инструмент. Фактически, это встроенный профилировщик systemd.
- Постарайтесь не ошибиться, интерпретируя вывод профилировщика!
- Всего два небольших изменения могут ускорить загрузку системы на семь секунд.
- Программное обеспечение, не способное работать с динамически меняющимся набором устройств, создает проблемы, и должно быть исправлено.
- Принудительное использование в стандартной установке Fedora 15 промышленной реализации подсистем хранения, возможно, является не самым правильным решением.

# 8 Новые конфигурационные файлы

Одно из ключевых достоинств systemd — наличие полного набора программ, необходимых на ранних стадиях загрузки, причем эти программы написаны на простом, быстром, надежном и легко поддающемся распараллеливанию языке C. Теперь можно отказаться от «простыней» shell-скриптов, разработанных для этих задач различными дистрибутивами. Наш «Проект нулевой оболочки»[25] увенчался полным успехом. Уже сейчас возможности предоставляемого нами инструментария покрывают практически все нужды настольных и встраиваемых систем, а также бóльшую часть потребностей серверов:

- Проверка и монтирование всех файловых систем.
- Обновление и активация квот на всех файловых системах.
- Установка имени хоста.
- Настройка сетевого интерфейса обратной петли (`lo`).
- Подгрузка правил SELinux, обновление меток безопасности в динамических каталогах `/run` и `/dev`.
- Регистрация в ядре дополнительных бинарных форматов (например, Java, Mono, WINE) через API-файловую систему `binfmt_misc`.
- Установка системной локали.
- Настройка шрифта и раскладки клавиатуры в консоли.
- Создание, очистка, удаление временных файлов и каталогов.
- Применение предписанных в `/etc/fstab` опций к смонтированным ранее API-файловым системам.
- Применение настроек `sysctl`.

[25] Наш девиз — «Первой оболочкой, запускающейся при старте системы, должна быть GNOME shell». Формулировка оставляет желать лучшего, но все же неплохо передает основную идею.

- Поддержка технологии упреждающего чтения (read ahead), включая автоматический сбор информации.
- Обновление записей в `utmp` при включении и выключении системы.
- Сохранение и восстановление затравки для генерации случайных чисел (random seed).
- Принудительная загрузка указанных модулей ядра.
- Поддержка шифрованных дисков и разделов.
- Автоматический запуск `getty` на serial-консолях.
- Взаимодействие с Plymouth.
- Создание уникального идентификатора системы.
- Настройка часового пояса.

В стандартной установке Fedora 15 запуск shell-скриптов требуется только для некоторых устаревших служб, а также для подсистемы хранения данных (поддержка LVM, RAID и multipath). Если они вам не нужны, вы легко можете отключить их, и наслаждаться загрузкой, полностью очищенной от shell-костылей (лично я это сделал уже давно). Такая загрузка является уникальной возможностью Linux-систем.

Большинство перечисленных выше компонентов настраиваются через конфигурационные файлы в каталоге `/etc`. Некоторые из этих файлов стандартизированы для всех дистрибутивов, и поэтому реализация их поддержки в наших инструментах не представляла особого труда. Например, это относится к файлам `/etc/fstab`, `/etc/crypttab`, `/etc/sysctl.conf`. Однако множество других, нестандартно расположенных файлов и каталогов вынуждали нас добавлять в код огромное количество операторов `#ifdef`, чтобы обеспечить поддержку различных вариантов расположения конфигураций в разных дистрибутивах. Такое положение дел сильно усложняет жизнь нам всем, и при этом ничем не оправдано — все эти файлы решают одни и те же задачи, просто немного по-разному.

Чтобы улучшить ситуацию и установить единый стандарт расположения базовых конфигурационных файлов во всех дистрибутивах, мы заставили systemd пользоваться дистрибутивно-специфическими конфигурациями только в качестве *резервного* варианта — основным источником информации становится определенный нами стандартный набор конфигурационных файлов. Разумеется, там, где это возможно, мы старались не придумывать чего-то принципиально нового, а брали лучшее из решений, предложенных существующими дистрибутивами. Ниже приводится небольшой обзор этого нового набора конфигурационных файлов, поддерживаемых systemd во всех дистрибутивах:

- `/etc/hostname`: имя хоста для данной системы. Одна из наиболее простых и важных системных настроек. В разных дистрибутивах оно настраивалось по-разному: Fedora использовала `/etc/sysconfig/network`, OpenSUSE — `/etc/HOSTNAME`, Debian — `/etc/hostname`. Мы остановились на варианте, предложенном Debian.
- `/etc/vconsole.conf`: конфигурация раскладки клавиатуры и шрифта для консоли.
- `/etc/locale.conf`: конфигурация общесистемной локали.
- `/etc/modules-load.d/*.conf`: каталог[26] для перечисления модулей ядра, которые нужно принудительно подгрузить при загрузке (впрочем, необходимость в этом возникает достаточно редко).

[26] Прим. перев.: Для описания этого и трех последующих каталогов автор пользуется термином «drop-in directory». Данный термин означает каталог, в который можно поместить множество независимых файлов настроек, и при чтении конфигурации все эти файлы будут обработаны (впрочем, часто накладывается ограничение — обрабатываются только файлы с именами, соответствующими маске, обычно `*.conf`). Такой подход позволяет значительно упростить процесс как ручного, так и автоматического конфигурирования различных компонентов — для внесения изменений в настройки уже не нужно редактировать основной конфигурационный файл, достаточно лишь скопировать/переместить в нужный каталог небольшой файл с указанием специфичных параметров.

- `/etc/sysctl.d/*.conf`: каталог для задания параметров ядра (`sysctl`). Дополняет классический конфигурационный файл `/etc/sysctl.conf`.

- `/etc/tmpfiles.d/*.conf`: каталог для управления настройками временных файлов (systemd обеспечивает создание, очистку и удаление временных файлов и каталогов, как во время загрузки, так и во время работы системы).

- `/etc/binfmt.d/*.conf`: каталог для регистрации дополнительных бинарных форматов (например, форматов Java, Mono, WINE).

- `/etc/os-release`: стандарт для файла, обеспечивающего идентификацию дистрибутива и его версии. Сейчас различные дистрибутивы используют для этого разные файлы (например, `/etc/fedora-release` в Fedora), и поэтому для решения такой простой задачи, как вывод имени дистрибутива, необходимо использовать базу данных, содержащую перечень возможных названий файлов. Проект LSB попытался создать такой инструмент — `lsb_release` — однако реализация столь простой функции через скрипт на Python'е является не самым оптимальным решением. Чтобы исправить сложившуюся ситуацию, мы решили перейти к единому простому формату представления этой информации.

- `/etc/machine-id`: файл с идентификатором данного компьютера (перекрывает аналогичный идентификатор D-Bus). Гарантируется, что в любой системе, использующей systemd, этот файл будет существовать и содержать корректную информацию (если его нет, он автоматически создается при загрузке). Мы вынесли этот файл из-под эгиды D-Bus, чтобы упростить решение множества задач, требующих наличия уникального и постоянного идентификатора компьютера.

- `/etc/machine-info`: новый конфигурационный файл, хранящий информации о полном (описательном) имени хоста (например, «Компьютер Леннарта») и значке, которым он будет обозначаться в графических оболочках, работающих с сетью (раньше этот значок мог определяться, например, файлом `/etc/favicon.png`). Данный конфигурационный файл обслуживается демоном `systemd-hostnamed`.

Одна из важнейших для нас задач — убедить *вас* использовать эти новые конфигурационные файлы в ваших инструментах для настройки системы. Если ваши конфигурационные фронтенды будут использовать новые файлы, а не их старые аналоги, это значительно облегчит портирование таких фронтендов между дистрибутивами, и вы внесете свой вклад в стандартизацию Linux. В конечном счете это упростит жизнь и администраторам, и пользователям. Разумеется, на текущий момент эти файлы полностью поддерживаются только дистрибутивами, основанными на systemd, но уже сейчас в их число входят практически все ключевые дистрибутивы, за исключением одного[27]. В этом есть что-то от «проблемы курицы и яйца»: стандарт становится настоящим стандартом только тогда, когда ему начинают следовать. В будущем мы намерены аккуратно форсировать процесс перехода на новые конфигурационные файлы: поддержка старых файлов будет удалена из systemd. Разумеется, процесс будет идти медленно, шаг за шагом. Но конечной его целью является переход всех дистрибутивов на единый набор базовых конфигурационных файлов.

Многие из этих файлов используются не только программами для настройки системы, но и апстримными проектами. Например, мы предлагаем проектам Mono, Java, WINE и другим помещать конфигурацию для регистрации своих бинарных форматов в `/etc/binfmt.d/` средствами их собственной сборочной системы. Специфичные для дистрибутивов механизмы поддержки бинарных форматов больше не нужны, и ваш проект будет работать одинаково хорошо во всех дистрибутивах. Аналогичное предложение мы обращаем и ко всем разработчикам программ, которым требуется автоматическое создание/очистка временных файлов и каталогов, например, в каталоге `/run`

[27] Прим. перев.: В конце 2010 года энтузиаст Andrew Edmunds добавил в systemd базовую поддержку Ubuntu и подготовил соответствующие пакеты, однако его инициатива не встретила поддержки среди менеджеров Canonical. На момент написания этих строк проект остается заброшенным с декабря 2010 г.

(ранее известном как `/var/run`). Таким проектам достаточно просто поместить соответствующий конфигурационный файл в `/etc/tmpfiles.d/`, тоже средствами собственной сборочной системы. Помимо прочего, подобный подход позволит увеличить скорость загрузки, так как, в отличие от SysV, не требует множества shell-скриптов, выполняющих тривиальные задачи (регистрация бинарных форматов, удаление/создание временных файлов/каталогов и т.п.). И пример того случая, когда апстримная поддержка стандартной конфигурации дала бы огромные преимущества — X11 (и его аналоги) могли бы устанавливать раскладку клавиатуры на основании данных из `/etc/vconsole.conf`.

Разумеется, я понимаю, что отнюдь не всех полностью устроят выбранные нами имена и форматы конфигурационных файлов. Но нам все же нужно было что-то выбрать, и мы выбрали то, что должно устроить большинство людей. Форматы конфигурационных файлов максимально просты, и их можно легко читать и записывать даже из shell-скриптов. Да, `/etc/bikeshed.conf` могло бы быть неплохим именем для файла конфигурации![28]

**Помогите нам стандартизировать Linux! Используйте новые конфигурационные файлы! Поддерживайте их в апстриме, поддерживайте их во всех дистрибутивах!**

И если у вас возникнет такой вопрос: да, все эти файлы так или иначе обсуждались с разными разработчиками из различных дистрибутивов. И некоторые из разработчиков планируют обеспечить поддержку новой конфигурации даже в системах без systemd.

# 9 О судьбе /etc/sysconfig и /etc/default

В дистрибутивах, основанных на Red Hat и SUSE, это каталог называется `/etc/sysconfig`. В дистрибутивах на базе Debian, его зовут `/etc/default`. Во многих других дистрибутивах также присутствуют каталоги похожего назначения. Связанные с ними вопросы неоднократно появляются в дискуссиях пользователей и разработчиков systemd. В этой статье мне хотелось бы рассказать, что я, как разработчик systemd, думаю об этих каталогах, и пояснить, почему, на мой взгляд, от них лучше отказаться. Стоит отметить, что это мое личное мнение, и оно может не совпадать с позицией проекта Fedora или моего работодателя.

Начнем с небольшого исторического экскурса. Каталог `/etc/sysconfig` появился в дистрибутивах Red Hat и SUSE задолго до того, как я присоединился к этим проектам — иными словами, это было очень давно. Некоторое время спустя, в Debian появился аналогичный по смыслу каталог `/etc/default`. Многие дистрибутивы используют такие каталоги, называя их по-разному. Они имеются даже в некоторых ОС семейства Unix. (Например, в SCO. Если эта тема вас заинтересовала — рекомендую обратиться к вашему знакомому ветерану Unix, он расскажет гораздо подробнее и интереснее, чем я.) Несмотря на то, что подобные каталоги широко используются в Linux и Unix, они совершенно не стандартизированы — ни в POSIX, ни в LSB/FHS, и результате мы имеем целый зоопарк их различных реализаций в разных дистрибутивах.

Назначение этих каталогов определено весьма расплывчато. Абсолютное большинство находящихся в них файлов являются включаемыми[29] shell-скриптами, содержащими, главным образом, определения переменных. Большинство файлов из этих каталогов включаются в одноименные скрипты SysV init. Данный принцип отражен в Debian Policy Manual (раздел 9.3.2) и в Fedora Packaging Guidelines, однако в обоих дистри-

[28] Прим. перев.: Здесь автор намекает на Паркинсоновский Закон Тривиальности, который гласит, что самые жаркие споры возникают вокруг наиболее простых вопросов. В частности, в качестве примера Паркинсон приводит обсуждение строительства атомной электростанции и гаража для велосипедов (bike shed) — если первое из этих решений принимается довольно быстро, то вокруг второго разгорается множество дискуссий по самым разным аспектам.

[29] Прим. перев.: Здесь автор использует термин sourcable, происходящий от bash'овской директивы `source`, обеспечивающей включение в скрипт кода из внешнего файла. В классическом POSIX shell это соответствует оператору-точке «`.`». В отличие от прямого запуска одного скрипта из другого, включаемый код исполняется той же самой оболочкой, что и основной код, и при возвращении в основной скрипт сохраняются переменные окружения, определенные во включаемом коде. Как правило, код для включения не содержит shebang'а (`#!/bin/sh` в начале файла).

бутивах иногда встречаются файлы, не соответствующие описанной схеме, например, не имеющие соответствующего init-скрипта, или даже сами не являющиеся скриптами.

Но почему вообще появились эти каталоги? Чтобы ответить на такой вопрос, обратимся к истории развития концепции SysV init-скриптов. Исторически, сложилось так, что они располагаются в каталоге под названием `/etc/rc.d/init.d` (или что-то похожее). Отметим, что каталог `/etc` вообще-то предназначен для хранения файлов конфигурации, а не исполняемого кода (в частности, скриптов). Однако, в начале своей истории, init-скрипты рассматривались именно как файлы конфигурации, и редактирование их администратором было общепринятой практикой. Но со временем, по мере роста и усложнения этих скриптов, их стали рассматривать уже не как файлы конфигурации, а как некие программы. Чтобы упростить их настройку и обеспечить безопасность процесса обновления, настройки были вынесены в отдельные файлы, загружаемые при работе init-скриптов.

Попробуем составить некоторое представление о настройках, которые можно сделать через эти файлы. Вот краткий и неполный список различных параметров, которые могут быть заданы через переменные окружения в таких файлах (составлен мною по результатам исследования соответствующих каталогов в Fedora и Debian):

- Дополнительные параметры командной строки для бинарника демона.
- Настройки локали для демона.
- Тайм-аут остановки для демона.
- Режим остановки для демона.
- Общесистемные настройки, например, системная локаль, часовой пояс, параметры клавиатуры для консоли.
- Избыточная информация о системных настройках, например, указание, установлены ли аппаратные часы по Гринвичу или по местному времени.
- Списки правил брандмауэра, не являются скриптами (!).
- Привязка к процессорным ядрам для демона.
- Настройки, не относящиеся к процессу загрузки, например, информация по установке пакетов с новыми ядрами, конфигурация nspluginwrapper, разрешение на выполнение предварительного связывания (prelinking) библиотек.
- Указание, нужно ли запускать данную службу или нет.
- Настройки сети.
- Перечень модулей ядра, которые должны быть подгружены принудительно.
- Нужно ли отключать питание компьютера при остановке системы (`poweroff`) или нет (`halt`).
- Права доступа для файлов устройств (!).
- Описание соответствующей SysV службы.
- Идентификатор пользователя/группы, значение umask для демона.
- Ограничения по ресурсам для демона.
- Приоритет OOM killer'а для демона.

А теперь давайте ответим на вопрос: что же такого неправильного в `/etc/sysconfig` (`/etc/default`) и почему этим каталогам нет места в мире systemd?

- Прежде всего, утрачены основная цель и смысл существования таких каталогов: файлы конфигурации юнитов systemd не являются программами, в отличие от init-скриптов SysV. Эти файлы представляют собой простые, декларативные описания конкретных задач и функций, и обычно содержат не более шести строк. Они легко могут быть сгенерированы и проанализованы без использования Bourne shell. Их легко читать и понимать. Кроме того, их легко модифицировать: достаточно скопировать файл из `/lib/systemd/system` в `/etc/systemd/system`, после чего внести в созданную копию необходимые изменения (при этом можно быть уверенным, что изменения не будут затерты пакетным менеджером). Изначальная причина появления обсуждаемых каталогов — необходимость разделять код и параметры конфигурации — больше не существует, так как файлы описания юнитов не являются кодом. Проще говоря, обсуждаемые каталоги являются решением проблемы, которой уже не существует.

- Обсуждаемые каталоги и файлы в них очень сильно привязаны к специфике дистрибутивов. Мы же планируем, используя systemd, способствовать стандартизации и унификации дистрибутивов. В частности, одним из факторов такой стандартизации является рекомендация распространять соответствующие файлы конфигурации юнитов сразу с апстримным продуктом, а не возлагать эту работу на создателей пакетов, как это делалась во времена SysV. Так как расположение обсуждаемых каталогов и настраиваемые через них параметры сильно отличаются от дистрибутива к дистрибутиву, пытаться поддерживать их в апстримных файлах конфигурации юнитов просто бессмысленно. Хранение параметров конфигурации в этих каталогах — один из факторов, превращающих Linux в зоопарк несовместимых решений.

- Большинство настроек, задаваемых через эти каталоги, являются избыточными в мире systemd. Например, различные службы позволяют задать таким методом параметры исполнения процесса, в частности, идентификатор пользователя/группы, ограничения ресурсов, привязки к ядрам CPU, приоритет OOM killer'a. Однако, эти настройки поддерживаются лишь некоторыми init-скриптами, причем одна и та же настройка в различных скриптах может называться по-разному. С другой стороны, в мире systemd, все эти настройки доступны для всех служб без исключения, и всегда задаются одинаково, через одни и те же параметры конфигурационных файлов.

- Файлы конфигурации юнитов имеют множество удобных и простых в использовании настроек среды исполнения процесса, гораздо больше, чем могут предоставить файлы из `/etc/sysconfig`.

- Необходимость в некоторых из этих настроек весьма сомнительна. Например, возьмем вышеупомянутую возможность задавать идентификатор пользователя/группы для процесса. Эту задачу должен решать разработчик ПО или дистрибутива. Вмешательство администратора в данную настройку, как правило, лишено смысла — только разработчик располагает всей информацией, позволяющий предотвратить конфликты идентификаторов и имен пользователей и групп.

- Формат файлов, используемых для сохранения настроек, плохо подходит для данной задачи. Так как эти файлы, как правило, являются включаемыми shell-скриптами, ошибки при их чтении очень трудно отследить. Например, ошибка в имени переменной приведет к тому, что переменная не будет изменена, однако никакого предупреждения при этом не выводится.

- Кроме того, такая организация не исключает влияния конфигурационных параметров на среду исполнения: например, изменение переменных `IFS` и `LANG` может существенно повлиять на результат интерпретации init-скрипта.

- Интерпретация этих файлов требует запуска еще одного экземпляра оболочки, что приводит к задержкам при загрузке[30].

- Файлы из `/etc/sysconfig` часто пытаются использовать в качестве суррогатной замены файлов конфигурации для тех демонов, которые не имеют встроенной поддержки конфигурационных файлов. В частности, вводятся специальные переменные, позволяющие задать аргументы командной строки, используемые при запуске демона. Встроенная поддержка конфигурационных файлов является более удобной альтернативой такому подходу, ведь, глядя на ключи «`-k`», «`-a`», «`-f`», трудно догадаться об их назначении. Очень часто, из-за ограниченности словаря, на различных демонов одни и те же ключи действуют совершенно по-разному (для одного демона ключ «`-f`» содержит указание демонизироваться при запуске, в то время как для другого эта опция действует прямо противоположным образом.) В отличие от конфигурационных файлов, строка запуска не может включать полноценных комментариев.

- Некоторые из настроек, задаваемых в `/etc/sysconfig`, являются полностью избыточными. Например, во многие дистрибутивах подобным методом указывается, установлены ли аппаратные часы компьютера по Гринвичу, или по местному времени. Однако эта же настройка задается третьей строкой файла `/etc/adjtime`, поддерживаемого во всех дистрибутивах. Использование избыточного и не стандартизированного параметра конфигурации только добавляет путаницу и не несет никакой пользы.

- Многие файлы настроек из `/etc/sysconfig` позволяют отключать запуск соответствующей службы. Однако эта операция уже поддерживается штатно для всех служб, через команды `systemctl enable`/`disable` (или `chkconfig on`/`off`). Добавление дополнительного уровня настройки не приносит никакой пользы и лишь усложняет работу администратора.

- Что касается списка принудительно загружаемых модулей ядра — в настоящее время существуют куда более удобные пути для автоматической подгрузки модулей при загрузке системы. Например, многие модули автоматически подгружаются `udev`'ом при обнаружении соответствующего оборудования. Этот же принцип распространяется на ACPI и другие высокоуровневые технологии. Одно из немногих исключений из этого правила — к сожалению, в настоящее время не поддерживается автоматическая загрузка модулей на основании информации о возможностях процессора, однако это будет исправлено в ближайшем будущем[31]. В случае, если нужный вам модуль ядра все же не может быть подгружен автоматически, все равно существует гораздо более удобные методы указать его принудительную подгрузку — например, путем создания соответствующего файла в каталоге `/etc/modules-load.d/` (стандартный метод настройки принудительной подгрузки модулей).

- И наконец, хотелось бы отметить, что каталог `/etc` определен как место для хранения системных настроек («Host-specific system configuration», согласно FHS). Наличие внутри него подкаталога `sysconfig`, который тоже содержит системную конфигурацию, является очевидно избыточным.

Что же можно предложить в качестве современной, совместимой с systemd альтернативы настройке системы через файлы в этих каталогах? Ниже приведены несколько рекомендаций, как лучше поступить с задаваемыми таким образом параметрами конфигурации:

- Попробуйте просто отказаться от них. Если они полностью избыточны (например, настройка аппаратных часов на Гринвич/местное время), то убрать их будет

[30] Прим. перев.: Здесь автор несколько заблуждается. Скрипты, включенные через директиву `source`, исполняются тем же экземпляром оболочки, что и вызвавший их скрипт.

[31] Прим. перев.: Необходимые патчи уже приняты в ядро, и соответствующая функция поддерживается Linux, начиная с версии 3.3.

довольно легко (если не рассматривать вопросы обеспечения совместимости). Если аналогичные по смыслу опции штатно поддерживаются systemd, нет никакого смысла дублировать их где-то еще (перечень опций, которые можно задать для любой службы, приведен на страницах справки systemd.exec(5) и systemd.service(5).) Если же ваша настройка просто добавляет еще один уровень отключения запуска службы — не плодите лишние сущности, откажитесь от нее.

- Найдите для них более подходящее место. Например, в случае с некоторыми общесистемными настройками (такими, как локаль или часовой пояс), мы надеемся аккуратно подтолкнуть дистрибутивы в правильном направлении (см. предыдущую главу).
- Добавьте их поддержку в штатную систему настройки демона через собственные файлы конфигурации. К счастью, большинство служб, работающих в Linux, являются свободным программным обеспечением, так что сделать это довольно просто.

Существует лишь одна причина поддерживать файлы `/etc/sysconfig` еще некоторое время: необходимо обеспечить совместимость при обновлении. Тем не менее, как минимум в новых пакетах, от таких файлов лучше отказаться.

В том случае, если требование совместимости критично, вы можете задействовать эти конфигурационные файлы даже в том случае, если настраиваете службы через штатные unit-файлы systemd. Если ваш файл из `sysconfig` содержит лишь определения переменных, можно воспользоваться опцией `EnvironmentFile=-/etc/sysconfig/foobar` (подробнее об этой опции см. systemd.exec(5)), позволяющей прочитать из файла набор переменных окружения, который будет установлен при запуске службы. Если же для задания настроек вам необходим полноценный язык программирования — ничто не мешает им воспользоваться. Например, вы можете создайть в `/usr/lib/<your package>/` простой скрипт, который включает соответствующие файлы, а затем запускает бинарник демона через `exec`. После чего достаточно просто указать этот скрипт в опции `ExecStart=` вместо бинарника демона.

# 10 Экземпляры служб

Большинство служб в Linux/Unix являются одиночными (singleton): в каждый момент времени на данном хосте работает только один экземпляр службы. В качестве примера таких одиночных служб можно привести Syslogd, Postfix, Apache. Однако, существуют службы, запускающие по несколько экземпляров себя на одном хосте. Например, службы наподобие Dovecot IMAP запускают по одному экземпляру на каждый локальный порт и/или IP-адрес. Другой пример, который можно встретить практически во всех системах — *getty*, небольшая служба, запускающаяся на каждом TTY (от `tty1` до `tty6`). На некоторых серверах, в зависимости от сделанных администратором настроек или параметров загрузки, могут запускаться дополнительные экземпляры getty, для подключаемых к COM-портам терминалов или для консоли системы виртуализации. Еще один пример службы, работающей в нескольких экземплярах (по крайней мере, в мире systemd) — *fsck*, программа проверки файловой системы, которая запускается по одному экземпляру на каждое блочное устройство, требующее такой проверки. И наконец, стоит упомянуть службы с активацией в стиле inetd — при обращении через сокет, по одному экземпляру на каждое соединение. В этой статье я попытаюсь рассказать, как в systemd реализовано управление «многоэкземплярными» службами, и какие выгоды системный администратор может извлечь из этой возможности.

Если вы читали предыдущие статьи из этого цикла, вы, скорее всего, уже знаете, что службы systemd именуются по схеме *foobar*`.service`, где *foobar* — строка, идентифицирующая службу (проще говоря, ее имя), а `.service` — суффикс, присутствующий в именах всех файлов конфигурации служб. Сами эти файлы могут находиться в каталогах `/etc/systemd/systemd` и `/lib/systemd/system` (а также, возможно, и в других[32]). Для

[32]Прим. перев.: Перечень каталогов, в которых выполняется поиск общесистемных юнит-файлов, приведен на странице руководства systemd(1) (раздел «System unit directories»). Указанные выше каталоги

служб, работающих в нескольких экземплярах, эта схема становится немного сложнее: *foobar*`@`*quux*`.service`, где *foobar* — имя службы, общее для всех экземпляров, а *quux* — идентификатор конкретного экземпляра. Например, `serial-gett@ttyS2.service` — это служба getty для COM-порта, запущенная на `ttyS2`.

При необходимости, экземпляры служб можно легко создать динамически. Скажем, вы можете, безо всяких дополнительных настроек, запустить новый экземпляр getty на последовательном порту, просто выполнив `systemctl start` для нового экземпляра:

```
# systemctl start serial-getty@ttyUSB0.service
```

Получив такую команду, systemd сначала пытается найти файл конфигурации юнита с именем, точно соответствующим запрошенному. Если такой файл найти не удается (при работе с экземплярами служб обычно так и происходит), из имени файла удаляется идентификатор экземпляра, и полученное имя используется при поиске *шаблона* конфигурации. В нашем случае, если отсутствует файл с именем `serial-getty@ttyUSB0.service`, используется файл-шаблон под названием `serial-getty@.service`. Таким образом, для всех экземпляров данной службы, используется один и тот же шаблон конфигурации. В случае с getty для COM-портов, этот шаблон, поставляемый в комплекте с systemd (файл `/lib/systemd/system/serial-getty@.service`) выглядит примерно так:

```
[Unit]
Description=Serial Getty on %I
BindTo=dev-%i.device
After=dev-%i.device systemd-user-sessions.service

[Service]
ExecStart=-/sbin/agetty -s %I 115200,38400,9600
Restart=always
RestartSec=0
```

(Заметим, что приведенная здесь версия немного сокращена, по сравнению с реально используемой в systemd. Удалены не относящиеся к теме нашего обсуждения параметры конфигурации, обеспечивающие совместимость с SysV, очистку экрана и удаление предыдущих пользователей с текущего TTY. Если вам интересно, можете посмотреть полную версию.)

Этот файл похож на обычный файл конфигурации юнита, с единственным отличием: в нем используются спецификаторы %I и %i. В момент загрузки юнита, systemd заменяет эти спецификаторы на идентификатор экземпляра службы. В нашем случае, при обращении к экземпляру `serial-getty@ttyUSB0.service`, они заменяются на «`ttyUSB0`». Результат такой замены можно проверить, например, запросив состояние для нашей службы:

```
$ systemctl status serial-getty@ttyUSB0.service
serial-getty@ttyUSB0.service - Getty on ttyUSB0
          Loaded: loaded (/lib/systemd/system/serial-getty@.service; static)
          Active: active (running) since Mon, 26 Sep 2011 04:20:44 +0200; 2s ago
        Main PID: 5443 (agetty)
          CGroup: name=systemd:/system/getty@.service/ttyUSB0
                  └ 5443 /sbin/agetty -s ttyUSB0 115200,38400,9600
```

Собственно, это и есть ключевая идея организации экземпляров служб. Как видите, systemd предоставляет простой в использовании механизм шаблонов, позволяющих динамически создавать экземпляры служб. Добавим несколько дополнительных замечаний, позволяющих эффективно использовать этот механизм:

Вы можете создавать дополнительные экземпляры таких служб, просто добавляя символьные ссылки в каталоги `*.wants/`. Например, чтобы обеспечить запуск getty на ttyUSB0 при каждой загрузке, достаточно создать такую ссылку:

---

`/etc/systemd/systemd` и `/lib/systemd/system` соответствуют значениям по умолчанию для упомянутых там переменных pkg-config `systemdsystemconfdir` и `systemdsystemunitdir` соответственно.

```
# ln -s /lib/systemd/system/serial-getty@.service \
/etc/systemd/system/getty.target.wants/serial-getty@ttyUSB0.service
```

При этом файл конфигурации, на который указывает ссылка (в нашем случае `serial-getty@.service`), будет вызван с тем именем экземпляра, которое указанно в названии этой ссылки (в нашем случае — `ttyUSB0`).

Вы не сможете обратиться к юниту-шаблону без указания идентификатора экземпляра. В частности, команда `systemctl start serial-getty@.service` завершится ошибкой.

Иногда возникает необходимость отказаться от использования общего шаблона для конкретного экземпляра (т.е. конфигурация данного экземпляра настолько сильно отличается от параметров остальных экземпляров данной службы, что механизм шаблонов оказывается неэффективен). Специально для таких случаев, в systemd и заложен предварительный поиск файла с именем, точно соответствующим указанному (прежде чем использовать общий шаблон). Таким образом, вы можете поместить файл с именем, точно соответствующим полному титулу экземпляра, в каталог `/etc/systemd/system/` — и содержимое этого файла, при обращении к выбранному экземпляру, полностью перекроет все настройки, сделанные в общем шаблоне.

В приведенном выше файле, в некоторых местах используется спецификатор `%I`, а в других — `%i`. У вас может возникнуть закономерный вопрос — чем они отличаются? `%i` всегда точно соответствует идентификатору экземпляра, в то время, как `%I` соответствует неэкранированной (unescaped) версии этого идентификатора. Когда идентификатор не содержит спецсимволов (например, `ttyUSB0`), результат в обоих случаях одинаковый. Но имена устройств, например, содержат слеши («/»), которые не могут присутствовать в имени юнита (и в имени файла на Unix). Поэтому, перед использованием такого имени в качестве идентификатора устройства, оно должно быть экранировано — «/» заменяются на «-», а большинство других специальных символов (включая «-») заменяются последовательностями вида `\xAB`, где AB — ASCII-код символа, записанный в шестнадцатеричной системе счисления[33]. Например, чтобы обратиться к последовательному USB-порту по его адресу на шине, нам нужно использовать имя наподобие `serial/by-path/pci-0000:00:1d.0-usb-0:1.4:1.1-port0`. Экранированная версия этого имени — `serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0`. `%I` будет ссылаться на первую из этих строк, `%i` — на вторую. С практической точки зрения, это означает, что спецификатор `%i` можно использовать в том случае, когда надо сослаться на имена других юнитов, например, чтобы описать дополнительные зависимости (в случае с `serial-getty@.service`, этот спецификатор используется для ссылки на юнит `dev-%i.device`, соответствующий одноименному устройству). В то время как `%I` удобно использовать в командной строке (`ExecStart`) и для формирования читабельных строк описания. Рассмотрим работу этих принципов на примере нашего юнит-файла[34]:

[33] Согласен, этот алгоритм дает на выходе не очень читабельный результат. Но этим грешат практически все алгоритмы экранирования. В данном случае, был использован механизм экранирования из udev, с одним изменением. В конце концов, нам нужно было выбрать что-то. Если вы собираетесь комментировать наш алгоритм экранирования — пожалуйста, оставьте свой адрес, чтобы я мог приехать к вам и раскрасить ваш сарай для велосипедов в синий с желтыми полосами. Спасибо!

[34] Прим. перев.: как видно из нижеприведенного примера, в командной строке `systemctl` используется экранированное имя юнита, что создает определенные трудности даже при наличии в оболочке «умного» автодополнения. Однако, начиная с systemd v186, при работе с `systemctl` можно указывать неэкранированные имена юнитов.

```
# systemctl start 'serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0.service'
# systemctl status 'serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0.service'
serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0.service - Serial Getty on serial/by-path/pci-0000:00:1d.0-usb-0:1.4:1.1-port0
          Loaded: loaded (/lib/systemd/system/serial-getty@.service; static)
          Active: active (running) since Mon, 26 Sep 2011 05:08:52 +0200; 1s ago
        Main PID: 5788 (agetty)
          CGroup: name=systemd:/system/serial-getty@.service/serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0
                  └ 5788 /sbin/agetty -s serial/by-path/pci-0000:00:1d.0-usb-0:1.4:1.1-port0 115200 38400 9600
```

Как видите, в качестве идентификатора экземпляра используется экранированная версия, в то время как в строке описания и в командной строке `getty` фигурирует неэкранированный вариант, как и предполагалось.

(Небольшое замечание: помимо `%i` и `%I`, существует еще несколько спецификаторов, и большинство из них доступно и в обычных файлах конфигурации юнитов, а не только в шаблонах. Подробности можно посмотреть на странице руководства, содержащей полный перечень этих спецификаторов с краткими пояснениями.)

# 11 Службы с активацией в стиле inetd

В одной из предыдущих глав (гл. 3) я рассказывал, как можно преобразовать SysV init-скрипт в юнит-файл systemd. В этой главе мы рассмотрим, как провести аналогичное преобразование для служб inetd.

Начнем с небольшого экскурса в историю вопроса. Уже многие годы inetd считается одной из базовых служб Unix-систем. Работая как суперсервер, он слушает сетевые сокеты от имени различных служб, и активирует соответствующие службы при поступлении на эти сокеты входящих соединений. Таким образом, он обеспечивает механизм активации по запросу (on-demand activation). Подобный подход избавляет от необходимости держать постоянно работающими все серверные процессы, что позволяет поддерживать множество служб даже на системах с очень ограниченными ресурсами. В дистрибутивах Linux можно встретить несколько различных реализаций inetd. Наиболее популярные из них — BSD inetd и xinetd. Хотя inetd во многих дистрибутивах до сих пор устанавливается по умолчанию, сейчас он уже редко используется для запуска сетевых служб — теперь большинство из них запускаются автономно при загрузке системы (основной аргумент в пользу такой схемы — она позволяет обеспечить более высокую производительность служб).

Одной из ключевых возможностей systemd (а также launchd от Apple) является сокет-активация — тот же самый механизм, давным-давно реализованный inetd, однако в данном случае ключевые цели немного другие. Сокет-активация в стиле systemd прежде всего ориентирована на локальные сокеты (`AF_UNIX`), хотя поддерживаются также и сетевые сокеты (`AF_INET`). И более важное отличие — сокет-активация в systemd обеспечивает не только экономию системных ресурсов, но также и эффективную параллелизацию работы (так как она позволяет запускать клиентские и серверные процессы одновременно, непосредственно после создания сокета), упрощение процесса конфигурации (отпадает необходимость явно указывать зависимости между службами) и увеличение надежности (перезапуск службы, служебный или экстренный — в случае падения — не приводит к недоступности сокета). Тем не менее, systemd ничуть не хуже inetd может запускать службы в ответ на входящие сетевые соединения.

Любая сокет-активация требует соответствующей поддержки со стороны самой запускаемой службы. systemd предоставляет очень простой интерфейс, который может быть использован службами для обеспечения сокет-активации. В основе этого простого и минималистичного механизма лежит функция `sd_listen_fds()`. Однако, интерфейс, традиционно используемый в inetd, еще проще. Он позволяет передавать запускаемой службе только один сокет, который формируется из потоков STDIN и STDOUT запущенного процесса. Поддержка этого механизма также присутствует в systemd — для обеспечения совместимости со множеством служб, у которых сокет-активация реализована только в стиле inetd.

Прежде, чем мы перейдем к примерам, рассмотрим три различных схемы, использующих сокет-активацию:

1. **Сокет-активация, ориентированная на параллелизацию, упрощение и надежность:** сокеты создаются на ранних стадиях загрузки, и единственный экземпляр службы тут же начинает обслуживать все поступающие запросы. Такая схема подходит для часто используемых системных служб — очевидно, что такие службы лучше запускать как можно раньше, и по возможности параллельно. В качестве примера можно привести D-Bus и Syslog.

2. **Сокет-активация для одиночных служб:** сокеты создаются на ранних стадиях загрузки, и единственный экземпляр службы запускается при получении входящих запросов. Такая схема больше подходит для редко используемых служб, позволяя обеспечить экономию системных ресурсов и уменьшить время загрузки, просто отложив активацию службы до того момента, когда она действительно понадобится. Пример — CUPS.

3. **Сокет-активация для служб, запускающих по экземпляру на каждое соединение:** сокеты создаются на ранней стадии загрузки, и при каждом входящем соединении запускается экземпляр службы, которому передается сокет соединения (слушающий сокет при этом остается у суперсервера, inetd или systemd). Эта схема подходит для редко используемых служб, не критичных по производительности (каждый новый процесс занимает сравнительно немного ресурсов). Пример: SSH.

Описанные схемы отнюдь не эквивалентны с точки зрения производительности. Первая и вторая схема, после завершения запуска службы, обеспечивают точно такую же производительность, как и в случае с независимой (stand-alone) службой (т.е. службой, запущенной без использования суперсервера и сокет-активации), так как слушающий сокет передается непосредственно процессу службы, и дальнейшая обработка входящих соединений происходит точно так же, как и в независимой службе. В то же время, производительность третьей из предложенных схем порою оставляет желать лучшего: каждое входящее соединение порождает еще один процесс службы, что, при большом количестве соединений, может привести к значительному потреблению системных ресурсов. Впрочем, эта схема имеет и свои достоинства. В частности, обеспечивается эффективная изоляция обработки клиентских запросов и, кроме того, выбор такой схемы активации значительно упрощает процесс разработки службы.

В systemd наибольшее внимание уделяется первой из этих схем, однако остальные две тоже прекрасно поддерживаются. В свое время, я рассказывал, какие изменения в коде службы нужно сделать для обеспечения работы по второй схеме, на примере сервера CUPS. Что же касается inetd, то он предназначен прежде всего для работы по третьей схеме, хотя поддерживает и вторую (но не первую). Именно из-за специфики этой схемы inetd получил репутации «медленного» (что, на самом деле, немного несправедливо).

Итак, изучив теоретическую сторону вопроса, перейдем к практике и рассмотрим, как inetd-служба может быть интегрирована с механизмами сокет-активации systemd. В качестве примера возьмем SSH, известную и широко распространенную службу. На большинстве систем, где она используется, частота обращений к ней не превышает одного раза в час (как правило, она *значительно* меньше этой величины). SSH уже очень давно поддерживает сокет-активацию в стиле inetd, согласно третьей схеме. Так как необходимость в данной службе возникает сравнительно редко, и число одновременно работающих процессов обычно невелико, она хорошо подходит для использования по этой схеме. Перерасход системных ресурсов должен быть незначителен: большую часть времени такая служба фактически не выполняется и не тратит ресурсы. Когда кто-либо начинает сеанс удаленной работы, она запускается, и останавливается немедленно по завершении сеанса, освобождая ресурсы. Что ж, посмотрим, как в systemd можно воспользоваться режимом совместимости с inetd и обеспечить сокет-активацию SSH.

Так выглядит строка с конфигурацией службы SSH для классического inetd:

```
ssh stream tcp nowait root /usr/sbin/sshd sshd -i
```

Аналогичный фрагмент конфигурации для xinetd:

```
service ssh {
        socket_type = stream
        protocol = tcp
        wait = no
        user = root
        server = /usr/sbin/sshd
        server_args = -i
}
```

Бóльшая часть опций интуитивно понятна, кроме того, нетрудно заметить, что оба этих текста повествуют об одном и том же. Однако, некоторые моменты не вполне очевидны. Например, номер прослушиваемого порта (22) не указывается в явном виде. Он определяется путем поиска имени службы в базе данных `/etc/services`. Подобный подход был некогда весьма популярен в Unix, но сейчас он уже постепенно выходит из моды, и поэтому в современных версиях xinetd поддерживается возможность явного указания номера порта. Одна из наиболее интересных настроек названа не вполне очевидно — `nowait` (`wait=no`). Она определяет, будет ли служба работать по второй (`wait`) или третьей (`nowait`) схеме. И наконец, ключ `-i` активирует inetd-режим в SSH-сервере (без этого ключа он работает в независимом режиме).

На языке systemd эти фрагменты конфигурации превращаются в два файла. Первый из них, `sshd.socket`, содержит информацию о прослушиваемом сокете:

```
[Unit]
Description=SSH Socket for Per-Connection Servers

[Socket]
ListenStream=22
Accept=yes

[Install]
WantedBy=sockets.target
```

Смысл большинства опций вполне очевиден. Сделаю лишь пару замечаний. `Accept=yes` соответствует режиму `nowait`. Надеюсь, предложенное мною название более точно отражает смысл опции — в режиме `nowait` суперсервер сам вызывает `accept()` для слушающего сокета, в то время как в режиме `wait` эта работа ложится на процесс службы. Опция `WantedBy=sockets.target` обеспечивает активацию данного юнита в нужный момент при загрузке системы.

Второй из этих файлов — `sshd@.service`:

```
[Unit]
Description=SSH Per-Connection Server

[Service]
ExecStart=-/usr/sbin/sshd -i
StandardInput=socket
```

Большинство представленных здесь опций, как всегда, понятны интуитивно. Особое внимание стоит обратить лишь на строчку `StandardInput=socket`, которая, собственно, и включает для данной службы режим совместимости с inetd-активацией. Опция `StandardInput=` позволяет указать, куда будет подключен поток STDIN процесса данной службы (подробности см. на странице руководства). Задав для нее значение `socket`, мы обеспечиваем подключение этого потока к сокету соединения, как того и требует механизм inetd-активации. Заметим, что явно указывать опцию `StandardOutput=` в данном случае необязательно — она автоматически устанавливается в то же значение, что и `StandardInput`, если явно не указано что-то другое. Кроме того, можно отметить наличие «-» перед именем бинарного файла sshd. Таким образом мы указываем systemd игнорировать код выхода процесса sshd. По умолчанию, systemd сохраняет коды выхода для всех экземпляров службы, завершившихся с ошибкой (сбросить эту информацию можно командой `systemctl reset-failed`). SSH довольно часто завершается с ненулевым кодом выхода, и мы разрешаем systemd не регистрировать подобные «ошибки».

Служба `sshd@.service` предназначена для работы в виде независимых экземпляров (такие службы мы рассматривали в предыдущей главе 10). Для каждого входящего соединения systemd будет создавать свой экземпляр этой службы, причем идентификатор экземпляра формируется на основе реквизитов соединения.

Быть может, вы спросите: почему для настройки inetd-службы в systemd требуется два файла конфигурации, а не один? Отвечаем: чтобы избежать излишнего усложнения, мы обеспечиваем максимально прозрачную связь между работающими юнитами и

соответствующими им юнит-файлами. Такой подход позволяет независимо оперировать юнитом сокета и юнитами соответствующих служб при формировании графа зависимостей и при управлении юнитами. В частности, вы можете остановить (удалить) сокет, не затрагивая уже работающие экземпляры соответствующей службы, или остановить любой из этих экземпляров, не трогая другие экземпляры и сам сокет.

Посмотрим, как наш пример будет работать. После того, как мы поместим оба предложенных выше файла в каталог `/etc/systemd/system`, мы сможем включить сокет (то есть, обеспечить его активацию при каждой нормальной загрузке) и запустить его (то есть активировать в текущем сеансе работы):

```
# systemctl enable sshd.socket
ln -s '/etc/systemd/system/sshd.socket' '/etc/systemd/system/sockets.target.wants/sshd.socket'
# systemctl start sshd.socket
# systemctl status sshd.socket
sshd.socket - SSH Socket for Per-Connection Servers
          Loaded: loaded (/etc/systemd/system/sshd.socket; enabled)
          Active: active (listening) since Mon, 26 Sep 2011 20:24:31 +0200; 14s ago
        Accepted: 0; Connected: 0
          CGroup: name=systemd:/system/sshd.socket
```

Итак, наш сокет уже прослушивается, но входящих соединений на него пока не поступало (счетчик `Accepted:` показывает количество принятых соединений с момента создания сокета, счетчик `Connected:` — количество активных соединений на текущий момент).

Подключимся к нашему серверу с двух разных хостов, и посмотрим на список служб:

```
$ systemctl --full | grep ssh
sshd@172.31.0.52:22-172.31.0.4:47779.service  loaded active running       SSH Per-Connection Server
sshd@172.31.0.52:22-172.31.0.54:52985.service loaded active running       SSH Per-Connection Server
sshd.socket                                   loaded active listening     SSH Socket for Per-Connection Serv
```

Как и следовало ожидать, работают два экземпляра нашей службы, по одному на соединение, и их в названиях указаны IP-адреса и TCP-порты источника и получателя. (Заметим, что в случае с локальными сокетами типа `AF_UNIX` там были бы указаны идентификаторы процесса и пользователя, соответствующие подключенному клиенту.) Таким образом, мы можем независимо выслеживать и убивать отдельные экземпляры sshd (например, если нам нужно прервать конкретный удаленный сеанс):

```
# systemctl kill sshd@172.31.0.52:22-172.31.0.4:47779.service
```

Вот, пожалуй, и все, что вам нужно знать о портировании inetd-служб в systemd и дальнейшем их использовании.

Применительно к SSH, в большинстве случаев схема с inetd-активацией позволяет сэкономить системные ресурсы и поэтому оказывается более эффективным решением, чем использование обычного service-файла sshd, обеспечивающего запуск одиночной службы без использования сокет-активации (поставляется в составе пакета и может быть включен при необходимости). В ближайшее время я собираюсь направить соответствующий запрос относительно нашего пакета SSH в багтрекер Fedora.

В завершение нашей дискуссии, сравним возможности xinetd и systemd, и выясним, может ли systemd полностью заменить xinetd, или нет. Вкратце: systemd поддерживает далеко не все возможности xinetd и поэтому отнюдь не является его полноценной заменой на все случаи жизни. В частности, если вы заглянете в список параметров конфигурации xinetd, вы заметите, что далеко не все эти опции доступны в systemd. Например, в systemd нет и никогда не будет встроенных служб `echo`, `time`, `daytime`, `discard` и т.д. Кроме того, systemd не поддерживает TCPMUX и RPC. Впрочем, бóльшая часть этих опций уже не актуальна в наше время. Подавляющее большинство inetd-служб не используют эти опции (в частности, ни одна из имеющихся в Fedora xinetd-служб не требует поддержки перечисленных опций). Стоит отметить, что xinetd имеет некоторые интересные возможности, отсутствующие в systemd, например, списки контроля

доступа для IP-адресов (IP ACL). Однако, большинство администраторов, скорее всего, согласятся, что настройка брандмауэра является более эффективным решением подобных задач[35], а для ценителей устаревших технологий systemd предлагает поддержку tcpwrap. С другой стороны, systemd тоже предоставляет ряд возможностей, отсутствующих в xinetd, в частности, индивидуальный контроль над каждым экземпляром службы (см. выше), и внушительный набор настроек для контроля окружения, в котором запускаются экземпляры. Я надеюсь, что возможностей systemd должно быть достаточно для решения большинства задач, а в тех редких случаях, когда вам потребуются специфические опции xinetd — ничто не мешает вам запустить его в дополнение к systemd. Таким образом, уже сейчас в большинстве случаев xinetd можно выкинуть из числа обязательных системных компонентов. Можно сказать, что systemd не просто возвращает функциональность классического юниксового inetd, но еще и восстанавливает ее ключевую роль в Linux-системах.

Теперь, вооруженные этими знаниями, вы можете портировать свои службы с inetd на systemd. Но, конечно, будет лучше, если этим займутся разработчики из апстрима приложения, или сопровождающие вашего дистрибутива.

# 12 К вопросу о безопасности

Одно из важнейших достоинств Unix-систем — концепция разделения привилегий между различными компонентами ОС. В частности, службы обычно работают от имени специальных системных пользователей, имеющих ограниченные полномочия, что позволяет уменьшить ущерб для системы в случае взлома этих служб.

Однако, такой подход предоставляет лишь самую минимальную защиту, так как системные службы, хотя уже и не получают полномочий администратора (root), все равно имеют практически те же права, что и обычные пользователи. Чтобы обеспечить более эффективную защиту, нужно поставить более жесткие ограничения, отняв у служб некоторые привилегии, присущие обычным пользователям.

Такая возможность предоставляется системами мандатного контроля доступа (далее MAC, от Mandatory Access Control), например, SELinux. Если вам нужно обеспечить высокий уровень безопасности на своем сервере, то вам определенно стоит обратить свое внимание на SELinux. Что же касается systemd, то он предоставляет разработчикам и администраторам целый арсенал возможностей по ограничению локальных служб, и эти механизмы работают независимо от систем MAC. Таким образом, вне зависимости от того, смогли ли вы разобраться с SELinux — у вас появляется еще несколько инструментов, позволяющих повысить уровень безопасности.

В этой главе мы рассмотрим несколько таких опций, предоставляемых systemd, и обсудим вопросы их практического применения. Реализация этих опций основана на использовании ряда уникальных технологий безопасности, интегрированных в ядро Linux уже очень давно, но при этом практически неизвестных для большинства разработчиков. Мы постарались сделать соответствующие опции systemd максимально простыми в использовании, чтобы заинтересовать администраторов и апстримных разработчиков. Вот краткий перечень наиболее интересных возможностей[36]:

- Изолирование служб от сети

[35] Прим. перев.: Стоит отметить, что приведенный пример является не единственным случаем, когда возможности брандмауэра Linux дублируются опциями xinetd. Например, количество соединений с каждого хоста может быть ограничено критерием connlimit, а скорость поступления входящих соединений можно контролировать сочетанием критериев limit и conntrack (`ctstate NEW`). Критерий recent позволяет создать аналог простейшей IDS/IPS, реализованной механизмом SENSORS в xinetd. Кроме того, в ряде случаев возможности брандмауэра значительно превосходят функциональность xinetd. Например, критерий hashlimit, опять-таки в сочетании с conntrack, позволяет ограничить скорость поступления входящих соединений с каждого хоста (не путать с критерием connlimit, ограничивающим количество одновременно открытых соединений). Также стоит отметить, что интегрированная в Linux подсистема ipset гораздо лучше подходит для работы с большими списками разрешенных/запрещенных адресов, нежели встроенные механизмы xinetd.

[36] Прим. перев.: В приведенном здесь списке не упомянута встроенная в systemd поддержка фильтров seccomp, так как она была добавлена уже после написания исходной статьи.

- Предоставление службам независимых каталогов `/tmp`
- Ограничение доступа служб к отдельным каталогам
- Принудительное отключение полномочий (capabilities) для служб
- Запрет форка, ограничение на создание файлов
- Контроль доступа служб к файлам устройств

Все эти опции описаны в man-страницах systemd, главным образом, в systemd.exec(5). Если вам потребуются какие-либо уточнения, пожалуйста, обратитесь к этим страницам.

Все эти опции доступны на системах с systemd, вне зависимости от использования SELinux или любой другой реализации MAC.

Все эти опции не так уж и обременительны, и поэтому их разумнее будет использовать даже в тех случаях, когда явная необходимость в них, казалось бы, отсутствует. Например: даже если вы полагаете, что ваша служба никогда не пишет в каталог `/tmp`, и поэтому использование `PrivateTmp=yes` (см. ниже) вроде бы и не обязательно — лучше включить эту опцию, просто потому, что вы не можете знать наверняка, как будут вести себя используемые вами сторонние библиотеки (и плагины для них). В частности, вы никогда не узнаете, какие модули NSS могут быть включены в каждой конкретной системе, и пользуются ли они каталогом `/tmp`.

Мы надеемся, что перечисленные опции окажутся полезными как для администраторов, защищающих свои системы, так и для апстримных разработчиков, желающих сделать свои службы безопасными «из коробки». Мы настоятельно рекомендуем разработчикам использовать такие опции по умолчанию в апстримных service-файлах — это сравнительно несложно, но дает значительные преимущества в плане безопасности.

## 12.1 Изолирование служб от сети

Простая, но мощная опция, которой вы можете воспользоваться при настройке службы — `PrivateNetwork=`:

```
...
[Service]
ExecStart=...
PrivateNetwork=yes
...
```

Добавление этой строчки обеспечивает полную изоляцию от сети всех процессов данной службы. Они будут видеть лишь интерфейс обратной петли (`lo`), причем полностью изолированный от обратной петли основной системы. Чрезвычайно эффективная защита против сетевых атак.

**Предупреждение:** Некоторым службам сеть необходима для нормальной работы. Разумеется, никто и не говорит о том, чтобы применять `PrivateNetwork=yes` к сетевым службам, таким, как Apache. Однако даже те службы, которые не ориентированы на сетевое взаимодействие, могут нуждаться в сети для нормального функционирования, и порой эта потребность не вполне очевидна. Например, если ваша система хранит учетные записи пользователей в базе LDAP, для выполнения системных вызовов наподобие `getpwnam()` может потребоваться обращение к сети. Впрочем, даже в такой ситуации, как правило, можно использовать `PrivateNetwork=yes`, за исключением случаев, когда службе может потребоваться информация о пользователях с UID $\geq 1000$.

Если вас интересуют технические подробности: эта возможность реализована с использованием технологии сетевых пространств имен (network namespaces). При задействовании данной опции, для службы создается новое пространство имен, в котором настраивается только интерфейс обратной петли.

## 12.2 Предоставление службам независимых каталогов /tmp

Еще одна простая, но мощная опция настройки служб — PrivateTmp=:

```
...
[Service]
ExecStart=...
PrivateTmp=yes
...
```

При задействовании этой опции, служба получит свой собственный каталог /tmp, полностью изолированный от общесистемного /tmp. По давно сложившейся традиции, в Unix-системах каталог /tmp является общедоступным для всех локальных служб и пользователей. За все эти годы, он стал источником огромного количества проблем безопасности. Чаще всего встречаются атаки с использованием символьных ссылок (symlink attacks) и атаки на отказ в обслуживании (DoS attacks). Использование независимой версии данного каталога для каждой службы делает подобные уязвимости практически бесполезными.

Для релиза Fedora 17 утверждена инициатива, направленная на включение этой опции по умолчанию для большинства системных служб.

**Предупреждение:** Некоторые службы используют /tmp не по назначению, помещая туда сокеты IPC и другие подобные элементы, что само по себе уже является уязвимостью (хотя бы потому, что используемые при передаче информации файлы и каталоги должны иметь предсказуемые имена, что делает подобные службы уязвимыми к атакам через символьные ссылки и атакам на отказ в обслуживании). Подобные объекты лучше помещать в каталог /run, так как в нем присутствует строгое разделение доступа. В качестве примера такого некорректного использования /tmp можно привести X11, размещающий там свои коммуникационные сокеты (впрочем, в данном конкретном случае некоторые меры безопасности все же присутствуют: сокеты размещаются в защищенном подкаталоге, который создается на ранних стадиях загрузки). Разумеется, для служб, использующих /tmp в целях коммуникации, включение опции PrivateTmp=yes недопустимо. К счастью, подобных служб сейчас уже не так уж и много.

Эта опция использует технологию пространств имен файловых систем (filesystem namespaces), реализованную в Linux. При включении данной опции, для службы создается новое пространство имен, отличающееся от иерархии каталогов основной системы только каталогом /tmp.

## 12.3 Ограничение доступа служб к отдельным каталогам

Используя опции ReadOnlyDirectories= и InaccessibleDirectories=, вы можете ограничить доступ службы к указанным каталогам только чтением, либо вообще запретить его:

```
...
[Service]
ExecStart=...
InaccessibleDirectories=/home
ReadOnlyDirectories=/var
...
```

Добавление этих двух строчек в файл конфигурации приводит к тому, что служба полностью утрачивает доступ к содержимому каталога /home (она видит лишь пустой каталог с правами доступа 000), а также не может писать внутрь каталога /var.

**Предупреждение:** К сожалению, в настоящее время опция ReadOnlyDirectories= не применяется рекурсивно к точкам монтирования, расположенным в поддереве указанного каталога (т.е. файловые системы, смонтированные в подкаталогах /var, попрежнему останутся доступными на запись, если, конечно, не перечислить их все поименно). Мы собираемся исправить это в ближайшее время.

Механизм работы этой опции тоже реализован с использованием пространств имен файловых систем.

## 12.4 Принудительное отключение полномочий (capabilities) для служб

Еще одна чрезвычайно эффективная опция — `CapabilityBoundingSet=`, позволяющая контролировать список capabilities, которые смогут получить процессы службы:

```
...
[Service]
ExecStart=...
CapabilityBoundingSet=CAP_CHOWN CAP_KILL
...
```

В нашем примере, служба может иметь лишь полномочия `CAP_CHOWN` и `CAP_KILL`. Попытки какого-либо из процессов службы получить любые другие полномочия, даже с использованием suid-бинарников, будут пресекаться. Список возможных полномочий приведен на странице документации capabilities(7). К сожалению, некоторые из них являются слишком общими (разрешают очень многое), например, `CAP_SYS_ADMIN`, однако выборочное делегирование полномочий все же является неплохой альтернативой запуску службы с полными административными (рутовыми) привилегиями.

Как правило, определение списка полномочий, необходимых для работы конкретной службы, является довольно трудоемким процессом, требующим ряда проверок. Чтобы немного упростить эту задачу, мы добавили возможность создания «черного списка» привилегий, как альтернативы описанному выше механизму «белого списка». Вместо того, чтобы указывать, какие привилегии можно оставить, вы можете перечислить, какие из них оставлять точно нельзя. Например: привилегия `CAP_SYS_PTRACE`, предназначенная для отладчиков, дает очень широкие полномочия в отношении всех процессов системы. Очевидно, что такие службы, как Apache, не занимаются отладкой чужих процессов, и поэтому мы можем спокойно отнять у них данную привилегию:

```
...
[Service]
ExecStart=...
CapabilityBoundingSet=~CAP_SYS_PTRACE
...
```

Наличие символа `~` после знака равенства инвертирует принцип работы опции: следующий за ним перечень привилегий рассматривается уже не как белый, а как черный список.

**Предупреждение:** Некоторые службы, при отсутствии определенных привилегий, могут вести себя некорректно. Как уже говорилось выше, формирование списка необходимых полномочий для каждой конкретной службы может оказаться довольно трудной задачей, и лучше всего будет обратиться с соответствующим запросом к разработчикам службы.

**Предупреждение 2:** Привилегии — отнюдь не лекарство от всех бед. Чтобы они работали действительно эффективно, возможно, придется дополнить их другими методиками защиты.

Чтобы проверить, какие именно привилегии имеют сейчас ваши процессы, вы можете воспользоваться программой `pscap` из комплекта `libcap-ng-utils`.

Применение опции `CapabilityBoundingSet=` является простой, прозрачной и удобной альтернативой введению аналогичных ограничений через модификацию исходного кода всех системных служб.

## 12.5 Запрет форка, ограничение на создание файлов

Некоторые меры безопасности можно ввести при помощи механизма ограничения ресурсов. Как следует из его названия, этот механизм предназначен прежде всего для контроля использования ресурсов, а не для контроля доступа. Однако, две его настройки могут быть использованы для запрета определенных действий: с помощью `RLIMIT_NPROC` можно запретить службе форкаться (запускать дополнительные процессы), а `RLIMIT_FSIZE` позволяет блокировать запись в файлы ненулевого размера:

```
...
[Service]
ExecStart=...
LimitNPROC=1
LimitFSIZE=0
...
```

Обратите внимание, что эти ограничения будут эффективно работать только в том случае, если служба запускается от имени простого пользователя (не root) и без привилегии `CAP_SYS_RESOURCE` (блокирование этой привилегии можно обеспечить, например, описанной выше опцией `CapabilityBoundingSet=`). В противном случае, ничто не мешает процессу поменять соответствующие ограничения.

**Предупреждение:** `LimitFSIZE=` действует очень жестко. Если процесс пытается записать файл ненулевого размера, он немедленно получает сигнал `SIGXFSZ`, который, как правило, прекращает работу процесса (в случае, если не назначен обработчик сигнала). Кроме того, стоит отметить, что эта опция не запрещает создание файлов нулевого размера.

Подробности об этих и других опциях ограничения ресурсов можно уточнить на странице документации setrlimit(2).

## 12.6 Контроль доступа служб к файлам устройств

Файлы устройств предоставляют приложениям интерфейс для доступа к ядру и драйверам. Причем драйверы, как правило, менее тщательно тестируются и не так аккуратно проверяются на предмет наличия уязвимостей, по сравнению с основными компонентами ядра. Именно поэтому драйверы часто становятся объектами атаки злоумышленников. systemd позволяет контролировать доступ к устройствам индивидуально для каждой службы:

```
...
[Service]
ExecStart=...
DeviceAllow=/dev/null rw
...
```

Приведенная конфигурация разрешит службе доступ только к файлу `/dev/null`, запретив доступ ко всем остальным файлам устройств.

Реализация данной опции построена на использовании cgroups-контроллера `devices`.

## 12.7 Прочие настройки

Помимо приведенных выше, простых и удобных в использовании опций, существует и ряд других других настроек, позволяющих повысить уровень безопасности. Но для их использования, как правило, уже требуются определенные изменения в исходном коде службы, и поэтому такие настройки представляют интерес прежде всего для апстримных разработчиков. В частности, сюда относится опция `RootDirectory=` (позволяющая запустить службу `chroot()`-окружении), а также параметры `User=` и `Group=`, определяющие пользователя и группу, от имени которых работает служба. Использование этих опций значительно упрощает написание демонов, так как работа по понижению привилегий ложится на плечи systemd.

Возможно, у вас возникнет вопрос, почему описанные выше опции не включены по умолчанию. Отвечаем: чтобы не нарушать совместимость. Многие из них несколько отступают от традиций, принятых в Unix. Например, исторически сложилось, так что `/tmp` является общим для всех процессов и пользователей. Существуют программы, использующие этот каталог для IPC, и включение по умолчанию режима изоляции для `/tmp` нарушит работу таких программ.

И несколько слов в заключение. Если вы занимаетесь сопровождением unit-файлов в апстримном проекте или в дистрибутиве, пожалуйста, подумайте о том, чтобы воспользоваться приведенными выше настройками. Если сопровождаемая вами служба станет более защищенной в конфигурации по умолчанию («из коробки»), от этого выиграют не только ваши пользователи, но и все мы, потому что Интернет станет чуть более безопасным.

# 13 Отчет о состоянии службы и ее журнал

При работе с системами, использующими systemd, одной из наиболее важных и часто используемых команд является `systemctl status`. Она выводит отчет о состоянии службы (или другого юнита). В таком отчете приводится статус службы (работает она или нет), список ее процессов и другая полезная информация.

В Fedora 17 мы ввели Journal, новую реализацию системного журнала, обеспечивающую структурированное, индексированное и надежное журналирование, при сохранении совместимости с классическими реализациями syslog. Собственно, мы начали работу над Journal только потому, что хотели добавить одну, казалось бы, простую, возможность: включить в вывод `systemctl status` последние 10 сообщений, поступивших от службы в системный журнал. Но на практике оказалось, что в рамках классических механизмов syslog, реализация этой возможности чрезвычайно сложна и малоэффективна. С другой стороны, сообщения службы в системный журнал несут очень важную информацию о ее состоянии, и такая возможность действительно была бы очень полезной, особенно при диагностике нештатных ситуаций.

Итак, мы интегрировали Journal в systemd, и научили `systemctl` работать с ним. Результат выглядит примерно так:

```
$ systemctl status avahi-daemon.service
avahi-daemon.service - Avahi mDNS/DNS-SD Stack
          Loaded: loaded (/usr/lib/systemd/system/avahi-daemon.service; enabled)
          Active: active (running) since Fri, 18 May 2012 12:27:37 +0200; 14s ago
        Main PID: 8216 (avahi-daemon)
          Status: "avahi-daemon 0.6.30 starting up."
          CGroup: name=systemd:/system/avahi-daemon.service
                  ├ 8216 avahi-daemon: running [omega.local]
                  └ 8217 avahi-daemon: chroot helper

May 18 12:27:37 omega avahi-daemon[8216]: Joining mDNS multicast group on interface eth1.IPv4 with address 1
May 18 12:27:37 omega avahi-daemon[8216]: New relevant interface eth1.IPv4 for mDNS.
May 18 12:27:37 omega avahi-daemon[8216]: Network interface enumeration completed.
May 18 12:27:37 omega avahi-daemon[8216]: Registering new address record for 192.168.122.1 on virbr0.IPv4.
May 18 12:27:37 omega avahi-daemon[8216]: Registering new address record for fd00::e269:95ff:fe87:e282 on et
May 18 12:27:37 omega avahi-daemon[8216]: Registering new address record for 172.31.0.52 on eth1.IPv4.
May 18 12:27:37 omega avahi-daemon[8216]: Registering HINFO record with values 'X86_64'/'LINUX'.
May 18 12:27:38 omega avahi-daemon[8216]: Server startup complete. Host name is omega.local. Local service c
May 18 12:27:38 omega avahi-daemon[8216]: Service "omega" (/services/ssh.service) successfully established.
May 18 12:27:38 omega avahi-daemon[8216]: Service "omega" (/services/sftp-ssh.service) successfully establis
```

Очевидно, это отчет о состоянии всеми любимого демона mDNS/DNS-SD, причем после списка процессов, как мы и обещали, приведены сообщения этого демона в системный журнал. Миссия выполнена!

Команда `systemctl status` поддерживает ряд опций, позволяющих настроить вывод информации в соответствии с вашими пожеланиями. Отметим две наиболее интересные из них: ключ `-f` позволяет в реальном времени отслеживать обновление сведений (по

аналогии с `tail -f`), а ключ `-n` задает количество выводимых журнальных записей (как нетрудно догадаться, по аналогии с `tail -n`).

Журнальные записи собираются из трех различных источников. Во-первых, это сообщения службы в системный журнал, отправленные через функцию libc `syslog()`. Во-вторых, это сообщения, отправленные через штатный API системы Journal. И наконец, это весь текст, выводимый демоном в STDOUT и STDERR. Проще говоря, все, что демон считает нужным сказать администратору, собирается, упорядочивается и отображается в одинаковом формате.

Добавленная нами возможность, при всей своей простоте, может оказаться чрезвычайно полезной практически любому администратору. По-хорошему, ее стоило реализовать еще 15 лет назад.

## 14 Самодокументированный процесс загрузки

Нам часто приходится слышать жалобы, что процесс загрузки при использовании systemd очень сложен для понимания. Не могу с этим согласиться. Более того, я берусь даже утверждать обратное: по сравнению с тем, что мы имели раньше (когда для того, чтобы разобраться в загрузке, нужно было иметь хорошие навыки программирования на языке Bourne Shell[37]), процесс загрузки стал более простым и прозрачным. Но определенная доля истины в этом критическом замечании все же есть: даже опытному Unix-администратору при переходе на systemd нужно изучить некоторые новые для себя вещи. Мы, разработчики systemd, обязаны максимально упростить такое обучение. Решая поставленную задачу, мы подготовили для вас кое-какие приятные сюрпризы, и предоставили хорошую документацию даже там, где это казалось невозможным.

Уже сейчас systemd располагает довольно обширной документацией, включая страницы руководства (на данный момент, их около сотни), wiki-сайт проекта, а также ряд статей в моем блоге. Тем не менее, большое количество документации еще не гарантирует простоты понимания. Огромные груды руководств выглядят пугающе, и неподготовленный читатель не может разобраться, с какого места ему начинать чтение, если его интересует просто общая концепция.

Чтобы решить данную проблему, мы добавили в systemd небольшую, но очень изящную возможность: самодокументированный[38] процесс загрузки. Что мы под этим подразумеваем? То, что для каждого элемента процесса загрузки теперь имеется документация, прозрачно привязанная к этому элементу, так что ее поиск не представляет трудности.

Иными словами, все штатные юниты systemd (которые, собственно, и формируют процесс загрузки, вызывая соответствующие программы) включают ссылки на страницы документации, описывающие назначение юнита/программы, используемые ими конфигурационные файлы и т.д. Если пользователь хочет узнать, для чего предназначен какой-либо юнит, какова его роль в процессе загрузки и как его можно настроить, может получить ссылки на соответствующую документацию, просто воспользовавшись давно известной командой `systemctl status`. Возьмем для примера `systemd-logind.service`:

```
$ systemctl status systemd-logind.service
systemd-logind.service - Login Service
          Loaded: loaded (/usr/lib/systemd/system/systemd-logind.service; static)
          Active: active (running) since Mon, 25 Jun 2012 22:39:24 -0200; 1 day and 18h ago
            Docs: man:systemd-logind.service(7)
                  man:logind.conf(5)
                  http://www.freedesktop.org/wiki/Software/systemd/multiseat
        Main PID: 562 (systemd-logind)
          CGroup: name=systemd:/system/systemd-logind.service
                  └ 562 /usr/lib/systemd/systemd-logind
```

[37] Чья привлекательность очень коварна и обманчива.

[38] Прим. перев.: В оригинале использован термин «self-explanatory», который также можно перевести как «само-объясняющий», «самоочевидный».

```
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event2 (Power Button)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event6 (Video Bus)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event0 (Lid Switch)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event1 (Sleep Button)
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/event7 (ThinkPad Extra Bu
Jun 25 22:39:25 epsilon systemd-logind[562]: New session 1 of user gdm.
Jun 25 22:39:25 epsilon systemd-logind[562]: Linked /tmp/.X11-unix/X0 to /run/user/42/X11-display.
Jun 25 22:39:32 epsilon systemd-logind[562]: New session 2 of user lennart.
Jun 25 22:39:32 epsilon systemd-logind[562]: Linked /tmp/.X11-unix/X0 to /run/user/500/X11-display.
Jun 25 22:39:54 epsilon systemd-logind[562]: Removed session 1.
```

На первый взгляд, в выводе этой команды почти ничего не изменилось. Но если вы вглядитесь повнимательнее, то заметите новое поле `Docs`, в котором приведены ссылки на документацию по данной службе. В нашем случае случае это два URI, ссылающихся на man-страницы, и один URL, указывающий на веб-страницу. man-страницы описывают соответственно предназначение службы и ее настройки, а веб-страница — базовые концепции, которые реализуются этой службой.

Тем, кто использует современные графические эмуляторы терминала, чтобы открыть соответствующую страницу документации, достаточно щелкнуть мышью по URI[39]. Иными словами, доступ к документации компонентов загрузки становится предельно простым: достаточно вызвать `systemctl status`, после чего щелкнуть по полученным ссылкам.

Если же вы используете не графический эмулятор терминала (где можно просто щелкнуть по URI), а настоящий терминал, наличие URI где-то в середине вывода `systemctl status` вряд ли покажется вам удобным. Чтобы упростить обращение к соответствующим страницам документации без использования мыши и копирования/вставки, мы ввели новую команду

```
systemctl help systemd-logind.service
```

которая просто откроет в вашем терминале соответствующие man-страницы.

URI, ссылающиеся на документацию, формируются в соответствии со страницей руководства uri(7). Поддерживаются схемы `http`/`https` (URL для веб-страниц) и `man`/`info` (локальные страницы руководства).

Разумеется, добавленная нами возможность не обеспечивает полной очевидности абсолютно всего, хотя бы потому, что пользователь должен как минимум знать про команду `systemctl status` (а также вообще про программу `systemctl` и про то, что такое юнит). Но при наличии этих базовых знаний, пользователю уже не составит труда получить информацию по интересующим его вопросам.

Мы надеемся, что добавленный нами механизм для связи работающего кода и соответствующей документации станет значительным шагом к полностью прозрачному и предельно простому для понимания процессу загрузки.

Описанная функциональность частично присутствует в Fedora 17, а в полном объеме она будет представлена в Fedora 18.

В завершение, стоит отметить, что использованная нами идея не является такой уж новой: в SMF, системе инициализации Solaris, уже используется практика указания ссылок на документацию в описании службы. Однако, в Linux такой подход является принципиально новым, и systemd сейчас является наиболее документированной и прозрачной системой загрузки для данной платформы.

Если вы занимаетесь разработкой или сопровождением пакетов и создаете файл конфигурации юнита, пожалуйста, включите в него ссылки на документацию. Это очень просто: достаточно добавить в секции `[Unit]` поле `Documentation=` и перечислить в нем соответствующие URI. Подробнее об этом поле см. на странице руководства systemd.unit(5). Чем больше будет документированных юнитов, тем проще станет работа системного администратора. (В частности, я уже направил в FPC предложение внести соответствующие пожелания в нормативные документы, регулирующие подготовку пакетов для Fedora.)

[39] Для нормальной работы данной функции, в эмуляторе терминала должен быть исправлена эта ошибка, а в программе просмотра страниц помощи — вот эта.

Да, кстати: если вас интересует общий обзор процесса загрузки systemd, то вам стоит обратить внимание на новую страницу руководства, где представлена псевгдорафическая потоковая диаграмма, описывающая процесс загрузки и роль ключевых юнитов.

# 15 Сторожевые таймеры

Разрабатывая systemd, мы ориентируемся на три основных области его применения: мобильные/встраиваемые устройства, настольные системы и промышленные серверы. Мобильные и встраиваемые системы, как правило, располагают очень скромными ресурсами и вынуждены очень экономно расходовать энергию. Настольные системы уже не так сильно ограничены по мощности, хотя все же проигрывают в этом плане промышленным серверам. Но, как ни странно, существуют возможности, востребованные в обоих крайних случаях (встраиваемые системы и серверы), но не очень актуальные в промежуточной ситуации (десктопы). В частности, к ним относится поддержка сторожевых таймеров (watchdogs), как аппаратных, так и программных.

Во встраиваемых системах часто используются аппаратные сторожевые таймеры, выполняющие сброс системы, когда программа перестает отвечать (точнее, перестает периодически посылать таймеру сигнал «все в порядке»). Таким образом, обеспечивается повышение надежности системы: что бы ни случилось, система обязательно попытается привести себя в рабочее состояние. На десктопах такая функциональность практически не востребована[40]. С другой стороны, она весьма актуальна для высокодоступных серверных систем.

Начиная с версии 183, systemd полностью поддерживает аппаратные сторожевые таймеры (доступные через интерфейс `/dev/watchdog`), а также обеспечивает программный сторожевой контроль системных служб. В целом эта схема (если она задействована) работает так. systemd периодически посылает сигналы аппаратному таймеру. В том случае, если systemd или ядро зависают, аппаратный таймер, не получив очередного сигнала, автоматически перезапустит систему. Таким образом, ядро и systemd защищены от бесконечного зависания — на аппаратном уровне. В то же время, сам systemd предоставляет интерфейс, реализующий логику программных сторожевых таймеров для отдельных служб. Таким образом можно обеспечить, например, принудительный перезапуск службы в случае ее зависания. Для каждой службы можно независимо настроить частоту опроса и задать соответствующее действие. Соединив оба звена (аппаратный сторожевой таймер, контролирующий ядро и systemd, и программные сторожевые таймеры, посредством которых systemd контролирует отдельные службы), мы получаем надежную схему надзора за всеми системными компонентами.

Чтобы задействовать аппаратный таймер, достаточно задать ненулевое значение параметра `RuntimeWatchdogSec=` в файле `/etc/systemd/system.conf`. По умолчанию этот параметр равен нулю (т.е. аппаратный таймер не задействован). Установив его равным, например, «`20s`», мы включим аппаратный сторожевой таймер. Если в течение 20 секунд таймер не получит очередного сигнала «все в порядке», система автоматически перезагрузится. Отметим, что systemd отправляет такие сигналы с периодом, равным половине заданного интервала — в нашем случае, через каждые 10 секунд. Собственно, это все. Просто задав один параметр, вы обеспечите аппаратный контроль работоспособности systemd и ядра[41].

Заметим, что с аппаратным сторожевым таймером (`/dev/watchdog`) должна работать только одна программа. Этой программой может быть либо systemd, либо отдельный демон сторожевого таймера (например, watchdogd) — выбор остается за вами.

Стоит упомянуть здесь еще одну опцию из файла `/etc/systemd/system.conf` — `ShutdownWatchdogSec=`. Она позволяет настроить аппаратный сторожевой таймер для контроля процесса перезагрузки. По умолчанию она равна «`10min`». Если в процессе

[40] Тем не менее, сейчас аппаратные сторожевые таймеры все чаще появляются и в настольных системах, так как стоят они относительно дешево и доступны практически во всех современных чипсетах.

[41] Небольшой совет: если вы занимаетесь отладкой базовых системных компонентов — не забудьте отключить сторожевой таймер. Иначе ваша система может неожиданно перезагрузиться, когда отладчик остановит процесс init и тот не может вовремя отправить сообщение таймеру.

остановки системы перед перезагрузкой она зависнет, аппаратный таймер принудительно перезагрузит ее по истечении данного интервала.

Это все, что я хотел сказать об аппаратных таймерах. Двух вышеописанных опций должно быть вполне достаточно для полноценного использования их возможностей. А сейчас мы рассмотрим логику программных сторожевых таймеров, обеспечивающих контроль работоспособности отдельных служб.

Прежде всего отметим, что для полноценной поддержки сторожевого контроля, программа должна содержать специальный код, периодически отправляющий таймеру сигналы «все в порядке». Добавить поддержку такой функциональности довольно просто. Для начала, демон должен проверить переменную окружения `WATCHDOG_USEC=`. Если она определена, то ее значение задает контрольный интервал в микросекундах, сформатированный в виде текстовой (ASCII) строки. В этом случае демон должен регулярно выполнять вызов `sd_notify("WATCHDOG=1")` с периодом, равным половине указанного интервала. Таким образом, поддержка программного сторожевого контроля со стороны демона сводится к проверке значения переменной окружения, и выполнении определенных действий в соответствии с этим значением.

Если интересующая вас служба обеспечивает поддержку такой функциональности, вы можете включить для нее сторожевой контроль, задав опцию `WatchdogSec=` в ее юнит-файле. Эта опция задает период работы таймера (подробнее см. systemd.service(5)). Если вы зададите ее, то systemd при запуске службы передаст ей соответствующее значение `WATCHDOG_USEC=` и, если служба перестанет своевременно отправлять сигналы «все в порядке», присвоит ей статус сбойной (failure state).

Очевидно, что одного только присвоения статуса недостаточно для обеспечения надежной работы системы. Поэтому нам также пригодятся настройки, определяющие, нужно ли перезапускать зависшую службу, количество попыток перезапуска, и дальнейшие действия, если она все равно продолжает сбоить. Чтобы включить автоматический перезапуск службы при сбое, нужно задать опцию `Restart=on-failure` в ее юнит-файле. Чтобы настроить, сколько раз systemd будет пытаться перезапустить службу, воспользуйтесь настройками `StartLimitBurst=` `StartLimitInterval=` (первая из них определяет предельное количество попыток, вторая — интервал времени, в течение которого они подсчитываются). В том случае, если достигнут предел количества попыток за указанное время, выполняется действие, заданное параметром `StartLimitAction=`. По умолчанию он установлен в `none` (никаких дополнительных действий не будет, службу просто оставят в покое со статусом сбойной). В качестве альтернативы можно указать одно из трех специальных действий: `reboot`, `reboot-force` и `reboot-immediate`. `reboot` соответствует обычной перезагрузке системы, с выполнением всех сопутствующих процедур (корректное завершение всех служб, отмонтирование файловых систем и т.д.). `reboot-force` действует более грубо — даже не пытаясь корректно остановить службы, оно просто убивает все их процессы, отмонтирует файловые системы и выполняет принудительную перезагрузку (в результате, перезагрузка происходит быстрее, чем обычно, но файловые системы остаются неповрежденными). И наконец, `reboot-immediate` даже не пытается отдать дань вежливости (убить процессы и отмонтировать файловый системы) — оно немедленно выполняет жесткую перезагрузку системы (это поведение практически аналогично срабатыванию аппаратного сторожевого таймера). Все перечисленный настройки подробно описаны на странице руководства systemd.service(5))[42].

Таким образом, мы мы получаем гибкий механизм для настройки сторожевого контроля служб, их перезапуска при зависании, и реагирования в ситуации, когда перезапуск не помогает.

Рассмотрим применение этих настроек на простом примере:

[42]Прим. перев.: Автор упускает из виду одну полезную опцию, непосредственно относящуюся к обсуждаемому вопросу — `OnFailure=`, задающую юнит, который будет активирован в случае сбоя исходного юнита. Таким образом можно обеспечить, например, запуск скриптов, отправляющих администратору уведомление о сбое в сочетании с дополнительной информацией (для сбора которой целесообразно задействовать команды `systemctl status` и `journalctl`). Кроме того, комбинирование данной настройки с опцией `OnFailureIsolate=` позволяет при сбое юнита перевести систему в определенное состояние (например, остановить некоторые службы, перейти в режим восстановления, выполнить перезагрузку).

```
[Unit]
Description=My Little Daemon
Documentation=man:mylittled(8)

[Service]
ExecStart=/usr/bin/mylittled
WatchdogSec=30s
Restart=on-failure
StartLimitInterval=5min
StartLimitBurst=4
StartLimitAction=reboot-force
```

Данная служба будет автоматически перезапущена, если она не передаст системному менеджеру очередной сигнал «все в порядке» в течение 30 секунд после предыдущего (кроме того, перезапуск будет произведен и в случае любого другого сбоя, например, завершения основного процесса службы с ненулевым кодом выхода). Если потребуется более 4 перезапусков службы за 5 минут — будет предпринято специальное действие, в данном случае, быстрая перезагрузка системы с корректным отмонтированием файловых систем.

Это все, что я хотел рассказать о сторожевых таймерах в systemd. Мы надеемся, что поддержки аппаратного отслеживания работоспособности процесса init, в сочетании с контролем функционирования отдельных служб, должно быть достаточно для решения большинства задач, связанных со сторожевыми таймерами. Разрабатываете ли вы встраиваемую либо мобильную систему, или работаете с высокодоступными серверами — вам определенно стоит попробовать наши решения!

(И да, если у вас возникнет вопрос, почему с аппаратным таймером должен работать именно init, и что мешает вынести эту логику в отдельный демон — пожалуйста, перечитайте эту главу еще раз, и попытайтесь увидеть всю цепочку сторожевого контроля как единое целое: аппаратный таймер надзирает за работой systemd, а тот, в свою очередь, следит за отдельными службами. Кроме того, мы полагаем, что отсутствие своевременного ответа от службы нужно рассматривать и обрабатывать так же, как и любые другие ее сбои. И наконец, взаимодействие с `/dev/watchdog` — одна из самых тривиальных задач в работе ОС (обычно, она сводится к простому вызову `ioctl()`), и для ее решения достаточно нескольких строк кода. С другой стороны, вынос данной функции в отдельный демон потребует организации сложного межпроцессного взаимодействия между этим демоном и процессом init — очевидно, реализация такой схемы предоставит значительно бóльший простор для ошибок, не говоря уже о повышенном потреблении ресурсов.)

Отметим, что встроенная в systemd поддержка аппаратного сторожевого таймера в конфигурации по умолчанию отключена, и поэтому никак не мешает работе с этим таймером из других программ. Вы без лишних проблем можете выбрать внешний сторожевой демон, если он лучше подходит для вашей задачи.

Да, и еще: если у вас возникнет вопрос, имеется ли в вашей системе аппаратный таймер — скорее всего да, если ваш компьютер не очень старый. Чтобы получить точный ответ, вы можете воспользоваться утилитой wdctl, включенной в последний релиз util-linux[43]. Эта программа выведет всю необходимую информацию о вашем аппаратном сторожевом таймере.

И в завершение, я хотел бы поблагодарить ребят из Pengutronix, которым принадлежит основная заслуга реализации сторожевого контроля в systemd.

[43]Прим. перев.: Утилита wdctl присутствует в util-linux, начиная с версии 2.22, которая, на момент написания этих строк, еще не вышла.

# 16 Запуск getty на последовательных (и не только) консолях

*Если вам лень читать всю статью целиком: для запуска getty на последовательной консоли*[44] *достаточно указать в загрузчике параметр ядра* `console=ttyS0`*, и systemd автоматически запустит getty на этом терминале.*

Физический последовательный порт RS-232, хотя уже и стал редкостью на современных настольных компьютерах, тем не менее, продолжает играть важную роль как на серверах, так и во встраиваемых системах. Он предоставляет простой и надежный доступ к управлению системой, даже когда сеть упала, а основной интерфейс управления завис. Кроме того, эмуляция последовательной консоли часто используется при управлении виртуальными машинами.

Разумеется, в Linux уже давно реализована поддержка работы с последовательными консолями но, при разработке systemd, мы постарались сделать работу с ними еще проще. В этой статье я хочу рассказать о том, как в systemd реализован запуск getty на терминалах различных типов.

Для начала, хотелось бы отметить один важный момент: в большинстве случаев, чтобы получить приглашение к логину на последовательном терминале, вам не нужно совершать никаких дополнительных действий: systemd сам проверит настройки ядра, определит их них используемую ядром консоль, и автоматически запустит на ней getty. Таким образом, вам достаточно лишь правильно указать ядру соответствующую консоль (например, добавив к параметрам ядра в загрузчик `console=ttyS0`).

Тем не менее, для общего образования все же стоит рассмотреть некоторые тонкости запуска getty в systemd. Эта задача решается двумя шаблонами юнитов[45]:

- `getty@.service` отвечает за виртуальные консоли (virtual terminals, VT, известные в системе под именами `/dev/tty1`, `/dev/tty2` и т.д.) — те, которые вы можете увидеть безо всякого дополнительного оборудования, просто переключившись на них из графического сеанса.

- `serial-getty@.service` обеспечивает поддержку всех прочих разновидностей терминалов, в том числе, подключаемых к последовательным портам (`/dev/ttyS0` и т.д.). Этот шаблон имеет ряд отличий от `getty@.service`, в частности, переменная `$TERM` в нем устанавливается в значение `vt102` (должно хорошо работать на большинстве физических терминалов), а не `linux` (которое работает правильно только на виртуальных консолях), а также пропущены настройки, касающиеся очистки буфера прокрутки (и поэтому имеющие смысл только на VT).

## 16.1 Виртуальные консоли

Рассмотрим механизм запуска `getty@.service`, обеспечивающий появление приглашений логина на виртуальных консолях (последовательны терминалы пока оставим в покое). По устоявшейся традиции, init-системы Linux обычно настраивались на запуск фиксированного числа экземпляров getty, как правило, шести (на первых шести виртуальных консолях, с `tty1` по `tty6`).

В systemd мы сделали этот процесс более динамичным: чтобы добиться большей скорости и эффективности, мы запускаем дополнительные экземпляры getty только при необходимости. Например, `getty@tty2.service` стартует только после того, как вы переключитесь на вторую виртуальную консоль. Отказавшись от обязательного запуска нескольких экземпляров getty, мы сэкономили немного системных ресурсов, а также

[44] Прим. перев.: Здесь и в дальнейшем автор использует термин «serial console». Точный перевод этого выражения на русский язык звучит как «консоль, подключаемая к последовательному порту». Однако, для краткости изложения, при переводе используется не вполне корректный, но хорошо знакомый администраторам жаргонизм «последовательная консоль». Также отметим, что в данном документе термины «консоль» и «терминал» используются как синонимы.

[45] Прим. перев.: Принципы работы с шаблонами и экземплярами служб изложены в главе 10. Для лучшего понимания нижеприведенного материала, рекомендуется перечитать эту главу, если вы успели ее подзабыть.

сделали загрузку системы чуть-чуть быстрее. При этом, с точки зрения пользователя, все осталось так же просто: как только он переключается на виртуальную консоль, на ней запускается getty, которая выводит приглашение к логину. Пользователь может и не подозревать о том, что до момента переключения приглашения не было. Тем не менее, если он войдет в систему и выполнит команду `ps`, он увидит, что getty запущены только на тех консолях, на которых он уже побывал.

По умолчанию, автоматический запуск getty производится на виртуальных консолях с первой по шестую (чтобы свести к минимуму отличия от привычной конфигурации)[46]. Отметим, что автоматический запуск getty на конкретной консоли производится только при условии, что эта консоль не занята другой программой. В частности, при интенсивном использовании механизма быстрого переключения пользователей графические сеансы могут занять первые несколько консолей (чтобы такое поведение не заблокировало возможность запуска getty, мы предусмотрели специальную защиту, см. чуть ниже).

Две консоли играют особую роль: `tty1` и `tty6`. `tty1`, при загрузке в графическом режиме, используется для запуска дисплейного менеджера, а при загрузке в многопользовательском текстовом режиме, systemd принудительно запускает на ней экземпляр getty, не дожидаясь переключений[47].

Что касается `tty6`, то она используется исключительно для автоматического запуска getty, и недоступна другим подсистемам, в частности, графическому серверу[48]. Мы сделали так специально, чтобы гарантировать возможность входа в систему в текстовом режиме, даже если графический сервер займет более пяти консолей.

## 16.2 Последовательные консоли

Работа с последовательными консолями (и всеми остальными видами невиртуальных консолей) реализована несколько иначе, чем с VT. По умолчанию, systemd запускает один экземпляр `serial-getty@.service` на основной консоли ядра[49] (если она не является виртуальной). Консолью ядра — это та консоль, на которую выводятся сообщения ядра. Обычно она настраивается в загрузчике, путем добавления к параметрам ядра аргумента наподобие `console=ttyS0`[50]. Таким образом, если пользователь перенаправил вывод ядра на последовательную консоль, то по завершении загрузки он увидит на этой консоли приглашение для логина[51]. Кроме того, systemd выполняет поиск консолей, предоставляемых системами виртуализации, и запускает `serial-getty@.service` на первой из этих консолей (`/dev/hvc0`, `/dev/xvc0` или `/dev/hvsi0`). Такое поведение реализовано специальной программой-генератором — systemd-getty-generator. Генераторы запускаются в самом начале загрузки и автоматически настраивают различные службы в зависимости от соответствующих факторов.

В большинстве случаев, вышеописанного механизма автоматической настройки должно быть достаточно, чтобы получить приглашение логина там, где нужно — без каких-либо дополнительных настроек systemd. Тем не менее, иногда возникает необходимость в ручной настройке — например, когда необходимо запустить getty сразу на нескольких последовательных консолях, или когда вывод сообщений ядра направляется на один терминал, а управление производится с другого. Для решения таких задач

[46] Тем не менее, это поведение можно легко изменить, задавая параметр `NAutoVTs=` в файле logind.conf.

[47] В данном случае нет принципиальной разницы между принудительным запуском и запуском по запросу: первая консоль используется по умолчанию, так что запрос на ее активацию обязательно поступит.

[48] При необходимости, вы можете легко поменять номер резервируемой консоли (или отключить резервирование), используя параметр `ReserveVT=` в файле logind.conf.

[49] Если для ядра настроен вывод в несколько консолей, *основной* считается та, которая идет *первой* в `/sys/class/tty/console/active`, т.е. указана *последней* в строке параметров ядра.

[50] Подробнее об этой опции см. в файле kernel-parameters.txt.

[51] Отметим, что getty, а точнее, `agetty` на такой консоли вызывается с параметром `-s`, и поэтому не изменяет настроек символьной скорость (baud rate) — сохраняется то значение, которое было указано в строке параметров ядра.

достаточно определить по экземпляру `serial-getty@.service` для каждого последовательного порта, на котором вы хотите запустить getty[52]:

```
# systemctl enable serial-getty@ttyS2.service
# systemctl start serial-getty@ttyS2.service
```

После выполнения этих команд, getty будет принудительно запускаться для указанных последовательных портов при всех последующих загрузках.

В некоторых ситуациях может возникнуть необходимость в тонкой настройке параметров getty (например, заданная для вывода сообщений ядра символьная скорость непригодна для интерактивного сеанса). Тогда просто скопируйте штатный шаблон юнита в каталог `/etc/systemd/system` и отредактируйте полученную копию:

```
# cp /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/serial-getty@ttyS2.service
# vi /etc/systemd/system/serial-getty@ttyS2.service
 ... редактируем параметры запуска agetty ...
# ln -s /etc/systemd/system/serial-getty@ttyS2.service /etc/systemd/system/getty.target.wants/
# systemctl daemon-reload
# systemctl start serial-getty@ttyS2.service
```

В приведенном примере создает файл настроек, определяющий запуск getty на порту `ttyS2` (это определяется именем, под которым мы скопировали файл — `serial-getty@ttyS2.service`). Все изменения настроек, сделанные в данном файле, будут распространяться только на этот порт.

Собственно, это все, что я хотел рассказать о последовательных портах, виртуальных консолях и запуске getty на них. Надеюсь, рассказ получился интересным.

# 17 Работа с Journal

В свое время, я уже рассказывал о некоторых возможностях journal (см. главу 13), доступных из утилиты `systemctl`. Сейчас я попробую рассказать о journal более подробно, с упором на практическое применение его возможностей.

Если вы еще не в курсе, что такое journal: это компонент systemd, регистрирующий сообщения из системного журнала (syslog), сообщения ядра (kernel log) и initrd, а также сообщения, которые процессы служб выводят на STDOUT и STDERR. Полученная информация индексируется и предоставляется пользователю по запросу. Journal может работать одновременно с традиционными демоном syslog (например, rsyslog или syslog-ng), либо полностью его заменять. Более подробно см. в первом анонсе.

Journal был включен в Fedora начиная с F17. В Fedora 18 journal вырос в мощный и удобный механизм работы с системным журналом. Однако, и в F17, и в F18 journal по умолчанию сохраняет информацию только в небольшой кольцевой буфер в каталоге `/run/log/journal`. Как и все содержимое каталога `/run`, эта информация теряется при перезагрузке[53]. Такой подход сильно ограничивает использование полезных возможностей journal, однако вполне достаточен для вывода актуальных сообщений от служб в `systemctl status`. Начиная с Fedora 19, мы собираемся включить сохранение логов на диск, в каталог `/var/log/journal`. При этом, логи смогут занимать гораздо больше места[54], а значит, смогут вместить больше полезной информации. Таким образом, journal станет еще более удобным.

[52] Отметим, что `systemctl enable` *для экземпляров служб* работает только начиная с systemd версии 188 и старше (например, в Fedora 18). В более ранних версиях придется напрямую манипулировать символьными ссылками: `ln -s /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/getty.target.wants/serial-getty@ttyS2.service ; systemctl daemon-reload`.

[53] Прим. перев.: Разумеется, это никак не относится к традиционному демону системного лога, даже если он работает поверх journal.

[54] Прим. перев.: В journal отдельно задаются ограничения на размер для логов во временном хранилище (`/run`) и в постоянном (`/var`). При превышении лимита старые журналы удаляются. Так как `/run` размещается на tmpfs, т.е. в оперативной памяти, для временного хранения по умолчанию установлены более жесткие ограничения. При необходимости, соответствующие настройки можно задать в файле journald.conf.

## 17.1 Сохранение логов на диск

В F17 и F18 вы можете включить сохранение логов на диск вручную:

```
# mkdir -p /var/log/journal
```

После этого рекомендуется перезагрузить систему, чтобы заполнить журнал новыми записями.

Так как теперь у вас есть journal, syslog вам больше не нужен (кроме ситуаций, когда вам совершенно необходимо иметь `/var/log/messages` в текстовом виде), и вы спокойно можете удалить его:

```
# yum remove rsyslog
```

## 17.2 Основы

Итак, приступим. Нижеприведенный текст демонстрирует возможности systemd 195, входящего в Fedora 18[55], так что, если некоторые из описанных трюков не сработают в F17 — пожалуйста, дождитесь F18.

Начнем с основ. Доступ к логам journal осуществляется через утилиту journalctl(1). Чтобы просто взглянуть на лог, достаточно ввести

```
# journalctl
```

Если вы выполнили эту команду с полномочиями root, вы увидите все журнальные сообщения, включая исходящие как от системных компонентов, так и от залогиненных пользователей[56]. Вывод этой команды форматируется в стиле `/var/log/messages`, но при этом добавлены кое-какие улучшения:

- Строки с приоритетом error и выше подсвечены красным.
- Строки с приоритетом notice и warning выделены жирным шрифтом.
- Все отметки времени сформированы с учетом вашего часового пояса.
- Для навигации по тексту используется просмотрщик (pager), по умолчанию `less`[57].
- Выводятся *все* доступные данные, включая информацию из файлов, прошедших ротацию (rotated logs).
- Загрузка системы отмечается специальной строкой, отделяющей записи, сгенерированные между (пере)загрузками.

Отметим, что в данной статье не приводятся примеры такого вывода — прежде всего, для краткости изложения, но также и для того, чтобы дать вам повод поскорее попробовать Fedora 18 с systemd 195. Надеюсь, отсутствие таких примеров не помешает вам уловить суть.

## 17.3 Контроль доступа

Итак, мы получили удобный и эффективный метод просмотра логов. Но для полного доступа к системным сообщениям требуются привилегии root, что не всегда удобно — в наше время администраторы предпочитают работать от имени непривилегированного

[55] Обновление со 195-й версией systemd на момент написания этих строк находится на тестировании и вскоре будет включено в состав Fedora 18.

[56] Прим. перев.: А если вы выполнили эту команду от имени непривилегированного пользователя, не входящего в группу `adm`, и при этом не включили сохранение логов на диск, то вы не увидите ничего — без специальных полномочий пользователь может просматривать только собственный лог, а он по умолчанию ведется только если логи записываются на диск.

[57] Прим. перев.: В инструментах systemd, включая journalctl, просмотрщик включается только при прямом выводе на экран, и отключается при перенаправлении вывода в файл или передаче его по каналу (shell pipe).

пользователя, переключаясь на root только при крайней необходимости. По умолчанию, непривилегированные пользователи могут просматривать в journal только свои собственные логи (сообщения, сгенерированные их процессами). Чтобы предоставить пользователю доступ ко всем системным логам, нужно включить его в группу `adm`:

```
# usermod -a -G adm lennart
```

Разлогинившись, а затем вновь залогинившись под именем `lennart`[58], я могу просматривать сообщения от всех пользователей и системных компонентов[59]:

```
$ journalctl
```

## 17.4 Отслеживание логов в реальном времени

Когда вы запускаете программу `journalctl` без параметров, она выводит все сообщения, сгенерированные на текущий момент, и возвращает управление оболочке. Однако, иногда бывает полезно отслеживать их появление в режиме реального времени. В классической реализации syslog это осуществлялось командой `tail -f /var/log/messages`. В journal ее аналог выглядит так:

```
$ journalctl -f
```

И работает он точно так же: выводит последние десять сообщений, после чего переходит в режим ожидания, и выводит новые сообщения по мере их появления.

## 17.5 Простейшие методы выборки записей

При вызове `journalctl` без параметров, она выводит все сообщения, начиная с самого первого из сохраненных. Разумеется, это огромный объем информации. На практике иногда бывает достаточно ограничиться сообщениями, сгенерированными с момента последней загрузки системы:

```
$ journalctl -b
```

Но часто даже после такой фильтрации записей остается довольно много. Что ж, мы можем ограничиться только наиболее важными. Итак, все сообщения с приоритетом error и выше, начиная с момента последней загрузки:

```
$ journalctl -b -p err
```

Если вы уже успели перезагрузить систему после того, как произошли интересующие вас события, целесообразнее будет воспользоваться выборкой по времени:

```
$ journalctl --since=yesterday
```

В результате мы увидим все сообщения, зарегистрированные начиная со вчерашнего дня вплоть до настоящего момента. Прекрасно! Разумеется, этот критерий отбора можно комбинировать с другими, например, с `-p err`. Но, допустим, нам нужно узнать о чем-то, что случилось либо 15-го октября, либо 16-го:

```
$ journalctl --since=2012-10-15 --until="2011-10-16 23:59:59"
```

Отлично, мы нашли то, что искали. Но вот вам сообщают, что сегодня ранним утром наблюдались проблемы с CGI-скриптами Apache. Ладно, послушаем, что нам скажет индеец:

```
$ journalctl -u httpd --since=00:00 --until=9:30
```

[58] Прим. перев.: Для того, чтобы обновить групповые полномочия в уже запущенных сеансах, можно воспользоваться командой newgrp(1): `newgrp adm`.

[59] Прим. перев.: Группа `adm` была выбрана на основании опыта дистрибутива Debian, в котором она устанавливается в качестве группы-владельца большинства лог-файлов. При этом авторы четко разделяют полномочия групп `adm` и `wheel`: если последняя используется для предоставления прав *изменять* что-либо в системе, то первая дает возможность лишь *просматривать* системную информацию.

Да, мы нашли это. Хм, похоже, что причиной стала проблема с диском `/dev/sdc`. Посмотрим, что с ним случилось:

```
$ journalctl /dev/sdc
```

Кошмар, ошибка ввода-вывода[60]! Нужно срочно заменить диск, пока не начались более серьезные проблемы. Ладно, пойдем дальше. Что у нас там случилось с процессом vpnc?

```
$ journalctl /usr/sbin/vpnc
```

Хм, ничего подозрительного. Но, кажется, проблема где-то во взаимодействии между `vpnc` и `dhclient`. Посмотрим объединенный и отсортированный по времени список сообщений от этих процессов:

```
$ journalctl /usr/sbin/vpnc /usr/sbin/dhclient
```

Отлично, мы нашли причину проблемы!

## 17.6 Продвинутые методы выборки

Да, это все, конечно, здорово, но попробуем подняться еще на ступеньку выше. Чтобы понять описанные ниже приемы, нужно знать, что systemd добавляет к каждой лог-записи ряд скрытых метаданных. Эти метаданные по структуре напоминают набор переменных окружения, хотя на самом деле дают даже больше возможностей: во-первых, метаданные могут включать большие бинарные блоки данных (впрочем, это скорее исключение — обычно они содержат текст в кодировке UTF-8), и во-вторых, в пределах одной записи поле метаданных может содержать сразу несколько значений (и это тоже встречается нечасто — обычно поля содержат по одному значению). Эти метаданные автоматически собираются и добавляются для каждой лог-записи, безо всякого участия пользователя. И вы легко можете их использовать для более тонкой выборки записей. Посмотрим, как они выглядят:

```
$ journalctl -o verbose -n1
Tue, 2012-10-23 23:51:38 CEST [s=ac9e9c423355411d87bf0ba1a9b424e8;i=4301;b=5335e9cf5d954633bb99ae
        PRIORITY=6
        SYSLOG_FACILITY=3
        _MACHINE_ID=a91663387a90b89f185d4e860000001a
        _HOSTNAME=epsilon
        _TRANSPORT=syslog
        SYSLOG_IDENTIFIER=avahi-daemon
        _COMM=avahi-daemon
        _EXE=/usr/sbin/avahi-daemon
        _SYSTEMD_CGROUP=/system/avahi-daemon.service
        _SYSTEMD_UNIT=avahi-daemon.service
        _SELINUX_CONTEXT=system_u:system_r:avahi_t:s0
        _UID=70
        _GID=70
        _CMDLINE=avahi-daemon: registering [epsilon.local]
        MESSAGE=Joining mDNS multicast group on interface wlan0.IPv4 with address 172.31.0.53.
        _BOOT_ID=5335e9cf5d954633bb99aefc0ec38c25
        _PID=27937
        SYSLOG_PID=27937
        _SOURCE_REALTIME_TIMESTAMP=1351029098747042
```

(Чтобы не утомлять вас огромным количеством текста, ограничимся одной записью. Ключ `-n` позволяет задать число выводимых записей, в нашем случае 1. Если указать его без параметра, будут показаны 10 последних записей.)

[60] Ну ладно, признаюсь, здесь я немножко считерил. Индексирование сообщений ядра по блочным устройствам пока что не принято в апстрим, но Ганс проделал огромную работу, чтобы реализовать эту функциональность, и я надеюсь, что к релизу F18 все будет.

Задав параметр `-o verbose`, мы переключили формат вывода — теперь, вместо скупых строчек, копирующих `/var/log/messages`, для каждой записи выводится полный перечень всех метаданных. В том числе, информация о пользователе и группе, контекст SELinux, идентификатор компьютера и т.д. Полный список общеизвестных полей метаданных приведен на соответствующей странице руководства.

База данных Journal индексируется по *всем* этим полям! И мы можем использовать любое из них в качестве критерия выборки:

```
$ journalctl _UID=70
```

Как нетрудно догадаться, в результате будут выведены все сообщения от процессов пользователя с UID 70. При необходимости, критерии можно комбинировать:

```
$ journalctl _UID=70 _UID=71
```

Указание нескольких значений для одного и того же поля эквивалентно логическому ИЛИ. Таким образом, будут выведены записи как от процессов с UID 70, так и от процессов с UID 71.

```
$ journalctl _HOSTNAME=epsilon _COMM=avahi-daemon
```

А вот указание нескольких *различных* полей дает эффект логического И. В результате, будут выведены записи только от процесса `avahi-daemon`, работающего на хосте с именем `epsilon`.

Но мы этим не ограничимся! Мы же суровые компьютерщики, нам нужны сложные логические выражения!

```
$ journalctl _HOSTNAME=theta _UID=70 + _HOSTNAME=epsilon _COMM=avahi-daemon
```

При помощи плюса мы можем явно задать логическое ИЛИ, применяя его к разным полям и даже И-выражениям. Поэтому наш пример выведет как записи с хоста `theta` от процессов с UID 70, так и с хоста `epsilon` от процесса `avahi-daemon`[61].

## 17.7 И немного магии

Уже неплохо, правда? Но есть одна проблема — мы же не сможем запомнить все возможные значения всех полей журнала! Для этого была бы нужна очень хорошая память. Но `journalctl` вновь приходит к нам на помощь:

```
$ journalctl -F _SYSTEMD_UNIT
```

Эта команда выведет все значения поля `_SYSTEMD_UNIT`, зарегистрированные в базе данных журнала на текущий момент. То есть, имена всех юнитов `systemd`, которые писали что-либо в журнал. Аналогичный запрос работает для всех полей, так что найти точное значение для выборки по нему — уже не проблема. Но тут самое время сообщить вам, что эта функциональность встроена в механизм автодополнения оболочки[62]! Это же просто прекрасно — вы можете просмотреть перечень значений поля и выбрать из него нужно прямо при вводе выражения. Возьмем для примера метки SELinux. Помнится, имя поле начиналось с букв SE...

```
$ journalctl _SE<TAB>
```

и оболочка сразу же дополнит:

```
$ journalctl _SELINUX_CONTEXT=
```

Отлично, и какое там значение нам нужно?

[61] Прим. перев.: Стоит отметить, что приоритет логических операций стандартный: сначала выполняются операции И, и только потом — операции ИЛИ. Используемая в `journalctl` система записи выражений аналогична принятой в классической алгебре: умножение (имеющее более высокий приоритет) не указывается знаком операции, а обозначается просто последовательной записью величин.

[62] Прим. перев.: В исходной статье речь идет только о bash, однако, начиная с релиза systemd 196, аналогичная функциональность доступна и для zsh.

```
$ journalctl _SELINUX_CONTEXT=<TAB><TAB>
kernel                                                  system_u:system_r:rtkit_daemon_t:s0
system_u:system_r:accountsd_t:s0                        system_u:system_r:syslogd_t:s0
system_u:system_r:avahi_t:s0                            system_u:system_r:system_cronjob_t:s0-s0:c0.c1023
system_u:system_r:bluetooth_t:s0                        system_u:system_r:system_dbusd_t:s0-s0:c0.c1023
system_u:system_r:chkpwd_t:s0-s0:c0.c1023               system_u:system_r:systemd_logind_t:s0
system_u:system_r:chronyd_t:s0                          system_u:system_r:systemd_tmpfiles_t:s0
system_u:system_r:crond_t:s0-s0:c0.c1023                system_u:system_r:udev_t:s0-s0:c0.c1023
system_u:system_r:devicekit_disk_t:s0                   system_u:system_r:virtd_t:s0-s0:c0.c1023 c0.c1023
system_u:system_r:dhcpc_t:s0                            system_u:system_r:vpnc_t:s0 sd_t:s0-s0:c0.c1023
system_u:system_r:dnsmasq_t:s0-s0:c0.c1023              system_u:system_r:xdm_t:s0-s0:c0.c1023
system_u:system_r:init_t:s0                             unconfined_u:system_r:rpm_t:s0-s0:c0.c1023
system_u:system_r:local_login_t:s0-s0:c0.c1023          unconfined_u:system_r:unconfined_t:s0-s0:c0.c1023
system_u:system_r:lvm_t:s0                              unconfined_u:system_r:useradd_t:s0-s0:c0.c1023
system_u:system_r:modemmanager_t:s0-s0:c0.c1023         unconfined_u:unconfined_r:unconfined_dbusd_t:s0-s0:c0.c1
system_u:system_r:NetworkManager_t:s0                   unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
system_u:system_r:policykit_t:s0
```

Ага, нас интересуют записи с меткой PolicyKit! Пользуясь дополнением, вводим:

```
$ journalctl _SELINUX_CONTEXT=system_u:system_r:policykit_t:s0
```

Очень просто, не правда ли! Пожалуй, самое простое из действий, связанных с SELinux ;-) Разумеется, такое дополнение работает для всех полей Journal.

На сегодня все. Впрочем, на странице руководства journalctl(1) можно узнать и о многих других возможностях, не описанных здесь. Например, о том, что `journalctl` может выводить данные в формате JSON, или в формате `/var/log/messages`, но с относительными метками времени, как в dmesg.

# 18 Управление ресурсами с помощью cgroups

Важную роль в современных компьютерных системах играют механизмы управления использованием ресурсов: когда вы запускаете на одной системе несколько программ, возникает необходимость распределять между ними ресурсы системы, в соответствии с некоторыми правилами. В частности, это особенно актуально на маломощных встраиваемых и мобильных системах, обладающих очень скудными ресурсами. Но та же задача актуальна и для очень мощных вычислительных кластеров, которые располагают огромными ресурсами, но при этом несут и огромную вычислительную нагрузку.

Исторически, в Linux поддерживался только одна схема управления ресурсами: все процессы получают примерно равные доли процессорного времени или потока ввода-вывода. При необходимости соотношение этих долей можно изменить при помощи значения *nice*, задаваемого для каждого процесса. Такой подход очень прост, и на протяжении долгих лет покрывал все нужды пользователей Linux. Но у него есть существенный недостаток: он оперирует лишь отдельными процессами, но не их группами. В результате, например, веб-сервер Apache с множеством CGI-процессов при прочих равных получает гораздо больше ресурсов, чем служба syslog, у которой не так много процессов.

В процессе проектирования архитектуры systemd, мы практически сразу поняли, что управление ресурсов должно быть одной из базовых функций, заложенных в основы его структуры. В современной системе — неважно, серверной или встраиваемой — контроль использования процессора, памяти и ввода-вывода для различных служб нельзя добавлять задним числом. Такая функциональность должна быть доступна изначально, через базовые настройки запуска служб. При этом, ресурсы должны распределяться на уровне служб, а не процессов, как это делалось при помощи значений nice или POSIX Resource Limits.

В этой статье я попробую рассказать о методах управления механизмами распределения ресурсов между службами systemd. Эта функциональность присутствует в systemd уже долгое время, и давно пора рассказать о ней пользователям и администраторам.

В свое время я пояснял[63], что контрольные группы Linux (cgroups) могут работать и как механизм группировки и отслеживания процессов, и как инструмент управления использованием ресурсов. Для функционирования systemd необходим только первый из этих режимов, а второй опционален. И именно этот опциональный второй режим дает вам возможность распределять ресурсы между службами. (А сейчас очень рекомендую вам, прежде чем продолжать чтение этой статьи, ознакомиться с базовой информацией о cgroups. Хотя дальнейшие рассуждения и не будут затрагивать низкоуровневые аспекты, все же будет лучше, если у вас сформируется некоторое представление о них.)

Основными контроллерами cgroups, отвечающими за управление ресурсами, являются cpu, memory и blkio. Для их использования необходимо, чтобы они были включены на этапе сборки ядра; большинство дистрибутивов (в том числе и Fedora), включают их в штатных ядрах. systemd предоставляет ряд высокоуровневых настроек, позволяющих использовать эти контроллеры, не вникая в технические детали их работы.

## 18.1 Процессор

Если в ядре включен контроллер `cpu`, systemd по умолчанию создает контрольную группу по этому ресурсу для каждой службы. Даже без каких-либо дополнительных настроек это дает положительных эффект: на системе под управлением systemd все службы получают равные доли процессорного времени, независимо от количества процессов, запущенных в рамках службы. Например, на вашем веб-сервере MySQL с несколькими рабочими процессами получит такую же долю процессорного времени, как и Apache, даже если тот запустил 1000 CGI-процессов. Разумеется, такое поведение при необходимости можно легко отключить — см. опцию `DefaultControllers=` в файле `/etc/systemd/system.conf`.

Если *равномерное* распределение процессорного времени между службами вас не устраивает, и вы хотите выделить определенным службам больше или меньше времени — используйте опцию `CPUShares=` в конфигурационном файле службы. По умолчанию это значение равно 1024. Увеличивая это число, вы даете службе больше процессорного времени, уменьшая — соответственно, меньше.

Рассмотрим небольшой практический пример. Допустим, вам нужно увеличить для службы Apache относительную долю потребления процессора до 1500. Для этого создаем файл «ручных» настроек `/etc/systemd/system/httpd.service`, который включает в себя все те же опции, что и файл настроек по умолчанию `/usr/lib/systemd/system/httpd.service`, отличаясь от него только значением `CPUShares=`:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
CPUShares=1500
```

Первая строка обеспечивает включение в нашу конфигурацию файла с настройками по умолчанию, сделанными разработчиками Apache или его сопровождающими в вашем дистрибутиве (если это включение не указать явно, данный файл будет проигнорирован). Далее, мы указываем тот параметр, который хотим изменить. Сохраняем файл,

[63] Прим. перев.: В указанном документе автор рассказывает, что контрольные группы Linux состоят из двух сущностей: **(A)** механизма иерархической группировки и маркировки процессов, и **(B)** механизма, позволяющего распределять ресурсы между полученными группами. Для работы (B) необходимо (A), но не наоборот — (A) может прекрасно работать без (B). Для нормально функционирования systemd (A) *необходим*, а (B) опционален (он лишь обеспечивает работу некоторых настроек). Вы можете собрать ядро только с необходимой для (A) опцией `CONFIG_CGROUPS=y`, отключив все связанные с (B) опции (такие как `CONFIG_CGROUP_FREEZER=y`, `CONFIG_CGROUP_DEVICE=y`, `CONFIG_CGROUP_CPUACCT=y`, `CONFIG_CGROUP_MEM_RES_CTLR=y`, `CONFIG_CGROUP_MEM_RES_CTLR_SWAP=y`, `CONFIG_CGROUP_MEM_RES_CTLR_KMEM=y`, `CONFIG_CGROUP_PERF=y`, `CONFIG_CGROUP_SCHED=y`, `CONFIG_BLK_CGROUP=y`, `CONFIG_NET_CLS_CGROUP=y`, `CONFIG_NET_PRIO_CGROUP=y`), и systemd будет нормально работать на такой системе (за исключением того, что связанные с этими контроллерами настройки не будут срабатывать). Однако, если собрать ядро без `CONFIG_CGROUPS=y`, функциональность systemd будет сильно ограничена. При этом, автор особо подчеркивает, что все негативные эффекты влияния контрольных групп на производительность обусловлены именно (B), в то время как (A) на производительность практически не влияет.

приказываем systemd перечитать конфигурацию, и перезапускаем Apache, чтобы настройки вступили в силу[64]:

```
systemctl daemon-reload
systemctl restart httpd.service
```

Готово!

Обратите внимание, что явное указание значения `CPUShares=` в конфигурации службы заставит systemd создать для нее контрольную группу в иерархии контроллера `cpu`, даже если этот контроллер не указан в `DefaultControllers=` (см. выше).

## 18.2 Отслеживание использования ресурсов

Для того, чтобы правильно распределять ресурсы между службами, неплохо бы знать реальные потребности этих служб. Чтобы упростить для вас отслеживание потребления ресурсов службами, мы подготовили утилиту systemd-cgtop, которая находит все имеющиеся в системе контрольные группы, определяет для каждой из них количество потребляемых ресурсов (процессорное время, память и ввод-вывод) и выводит эти данные в динамически обновляемой сводной таблице, по аналогии с программой top. Используя вводимое systemd распределение служб по контрольным группам, эта утилита выводит для служб те же сведения, которые top выводит для отдельных процессов.

К сожалению, по умолчанию `cgtop` может раздельно отслеживать для каждой службы только потребление процессорного времени, а сведения по использованию памяти и ввода-вывода доступны только для всей системы в целом. Это ограничение возникает из-за того, что в конфигурации по умолчанию контрольные группы для служб создаются только в иерархии контроллера `cpu`, но не `memory` и `blkio`. Без создания групп в иерархии этих контроллеров невозможно отследить использование ресурса по службам. Самый простой способ обойти это ограничение — приказать systemd создавать соответствующие группы, добавив `memory` и `blkio` в перечень `DefaultControllers=` в файле `system.conf`.

## 18.3 Память

Используя опции `MemoryLimit=` и `MemorySoftLimit=`, вы можете ограничивать суммарное потребление оперативной памяти всеми процессами службы. В них указывается предел потребления памяти в байтах[65]. При этом поддерживаются суффиксы K, M, G и T, обозначающие соответственно, килобайт, мегабайт, гигабайт и терабайт (по основанию 1024).

```
.include /usr/lib/systemd/system/httpd.service

[Service]
MemoryLimit=1G
```

По аналогии с `CPUShares=`, явное указание этих опций заставит systemd создать для службы контрольную группу в иерархии контроллера `memory`, даже если он не был указан в `DefaultControllers=`.

[64] Прим. перев.: systemd версий до 197 включительно, не поддерживает изменение параметров контрольных групп без перезапуска службы. Но вы можете узнать контрольную группу службы командой наподобие `systemctl show -p ControlGroup avahi-daemon.service`, и выполнить настройки любым удобным для вас способом, например, через запись значений в псевдофайлы cgroupfs (разумеется, при следующем запуске службы к ней будут применены параметры, указанные в конфигурационном файле). Начиная с systemd 198, в программу `systemctl` добавлена поддержка команд `set-cgroup-attr`, `unset-cgroup-attr` и `get-cgroup-attr`.

[65] Прим. перев.: Разница между `MemorySoftLimit=` и `MemoryLimit=` состоит в том, что первый предел можно превышать, если в системе еще есть достаточное количество свободной памяти. Второй из этих пределов превышать нельзя, независимо от наличия свободной памяти. Подробнее см. раздел «Soft limits» в файле документации.

## 18.4 Ввод-вывод

Для контроля пропускной полосы ввода-вывода с блочных устройств, доступно несколько настроек. Первая из них — `BlockIOWeight=`, задающая *долю* полосы ввода-вывода для указанной службы. Принцип похож на `CPUShares=` (см. выше), однако здесь величина относительной доли ограничена значениями от 10 до 1000. По умолчанию, она равна 1000. Уменьшить долю для службы Apache можно так:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOWeight=500
```

При необходимости, вы можете задать такое значение отдельно для каждого устройства:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOWeight=/dev/disk/by-id/ata-SAMSUNG_MMCRE28G8MXP-0VBL1_DC06K01009SE009B5252 750
```

При этом, точное название устройства знать не обязательно — достаточно указать интересующий вас каталог:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOWeight=/home/lennart 750
```

Если заданный вами путь не указывает на файл устройства, systemd автоматически определит, на каком устройстве расположен указанный файл/каталог, и выставит для этого устройства соответствующую настройку.

Вы можете добавить несколько таких строк, задавая долю пропускной полосы отдельно для различных устройств, и при этом также допускается указать «общее» значение (как в первом примере), которое будет использовано для всех остальных устройств.

В качестве альтернативы относительной доле пропускной полосы, вы также можете ограничивать абсолютную долю, используя настройки `BlockIOReadBandwidth=` и `BlockIOWriteBandwidth=`. В них нужно указать устройство или любой находящийся на нем файл/каталог, а также предельную скорость чтения/записи в байтах в секунду:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOReadBandwith=/var/log 5M
```

В результате, для данной службы скорость чтения с устройства, содержащего каталог `/var/log`, будет ограничена величиной 5 мегабайт в секунду.

По аналогии с вышеописанными `CPUShares=` и `MemoryLimit=`, явное указание любой из приведенных настроек пропускной полосы заставит systemd создать для службы контрольную группу в иерархии контроллера `blkio`.

## 18.5 Прочие параметры

Вышеописанные опции покрывают лишь малую толику настроек, поддерживаемых различными контроллерами Linux cgroups. Мы добавили высокоуровневый интерфейс только к тем настройкам, которые кажутся нам наиболее важным для большинства пользователей. Из соображений удобства мы добавили механизмы, обеспечивающие поддержку крупных единиц измерения (килобайты, мегабайты и т.д.) и автоматическое определение блочных устройств по указанному файлу/каталогу.

В некоторых случаях описанных высокоуровневых настроек может оказаться недостаточно — допустим, вам нужно задать низкоуровневую настройку cgroups, для которой мы (пока) не добавили высокоуровневого аналога. На этот случай мы предусмотрели универсальных механизм задания таких опций в конфигурационных файлах юнитов. Рассмотрим, например, задание для службы параметра *swappiness* (относительная интенсивность использования подкачки для процессов службы). В systemd нет высокоуровневой настройки для этого значения. Однако вы можете задать его, используя низкоуровневую настройку `ControlGroupAttribute=`:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
ControlGroupAttribute=memory.swappiness 70
```

Как обычно, явное указание настройки, относящейся к какому-либо контроллеру (в нашем случае `memory`) приведет к автоматическому созданию группы в иерархии данного контроллера.

В дальнейшем, возможно, мы расширим возможности высокоуровневой настройки различных параметров контрольных групп. Если вы часто пользуетесь какими-то из них и полагаете, что для них можно добавить соответствующие опции — не стесняйтесь обращаться к нам. А лучше всего — присылайте сразу патч!

**Предупреждение:** Обратите внимание, что использование некоторых контроллеров может сильно сказаться на производительности системы. Это та цена, которую приходится платить за контроль над ресурсами. Использование таких контроллеров может ощутимо замедлить некоторые операции. В частности, весьма нелестная в этом плане репутация закрепилась за контроллером `memory` (хотя, не исключено, что эта проблема уже исправлена в свежих выпусках ядра).

Для углубленного изучения темы, затронутой в этой статье, вы можете обратиться к документации по поддерживаемым настройкам юнитов, а также по контроллерам cpu, memory и blkio.

Стоит подчеркнуть, что мы сейчас обсуждали распределение ресурсов *между службами*. В дополнение к этим современным механизмам, systemd также поддерживает и традиционные настройки, касающиеся распределения ресурсов *между отдельными процессами*. Хотя такие настройки обычно наследуются порожденными процессами, они, тем не менее, все равно ограничивают ресурсы на уровне отдельных процессов. В частности, к ним относятся `IOSchedulingClass=`, `IOSchedulingPriority=`, `CPUSchedulingPolicy=`, `CPUSchedulingPriority=`, `CPUAffinity=`, `LimitCPU=` и т.п. Для их работы не требуют контроллеры cgroups, и они не так сильно ухудшают производительность. Возможно, мы рассмотрим их в последующих статьях.

# 19 Проверка на виртуальность

Еще в начале разработки systemd, мы внимательно изучали существовавшие на тот момент init-скрипты, выделяя наиболее типичные для них операции. Среди прочих, в составленный нами список попала и такая функция, как определение виртуализации: некоторые скрипты проверяли, запускаются они в виртуальном окружении (например, KVM, VMWare, LXC и т.д.) или на полноценной, физической системе. Часть этих скриптов отказывалась работать на виртуальных системах (например, службы управления устройствами совершенно излишни в виртуальных контейнерах, не имеющих доступа к устройствам), другие же, наоборот, запускались только в определенных виртуальных окружениях (например, всевозможные «guest additions», рекомендуемые к запуску на гостевых системах VMWare и VirtualBox). По-хорошему, в некоторых ситуациях было бы более правильно проверять некоторые другие условия, а не пытаться явно определить наличие виртуализации. Тем не менее, всесторонне изучив вопрос, мы пришли к выводу, что во многих случаях возможность явной проверки такого условия при запуске служб была бы очень кстати. В результате, мы добавили поддержку соответствующей

опции настройки юнитов — `ConditionVirtualization`; кроме того, мы создали небольшую утилиту, которую можно вызывать из скриптов — `systemd-detect-virt(1)`; и наконец, мы предоставили простой интерфейс для шины D-Bus, позволяющий получить информацию о виртуализации даже непривилегированным программам.

Определить, запущен код на виртуальной системе, или на физической, на самом деле не так уж и сложно. В зависимости от того, какие именно механизмы виртуализации вы хотите определить, основная работа сводится к выполнению инструкции CPUID и, возможно, проверке некоторых файлов в `/sys` и `/proc`. Основная трудность здесь — точно знать строки, которые нужно искать. Список таких строк необходимо поддерживать в актуальном состоянии. В настоящий момент, systemd определяет следующие механизмы виртуализации:

- Полная виртуализация (т.е. виртуальные машины):
  - qemu
  - kvm
  - vmware
  - microsoft
  - oracle
  - xen
  - bochs
- Виртуализация на уровне ОС (т.е. контейнеры):
  - chroot
  - openvz
  - lxc
  - lxc-libvirt
  - systemd-nspawn

Рассмотрим, как можно использовать эту функциональность.

## 19.1 Условия на запуск юнитов

При помощи опции `ConditionVirtualization`, добавленной в секцию `[Unit]` файла конфигурации юнита, вы можете обеспечить запуск (или наоборот, отмену запуска) данного юнита в зависимости от того, работает ли он на виртуальной системе, или нет. В случае утвердительного ответа, также можно уточнить, какая система виртуализации при этом используется. Например:

```
[Unit]
Name=My Foobar Service (runs only only on guests)
ConditionVirtualization=yes

[Service]
ExecStart=/usr/bin/foobard
```

Помимо «`yes`» или «`no`», вы также можете указать идентификатор конкретной системы виртуализации (согласно списку выше, например, «`kvm`», «`vmware`» и т.д.), либо «`container`» или «`vm`» (что позволит отличить виртуализацию на уровне ОС от полной виртуализации). Кроме того, вы можете добавить перед значением восклицательный знак, и результат проверки будет инвертирован (юнит запустится только в том случае, если указанная технология *не* используется). Подробности вы можете узнать на странице руководства.

### 19.2 В скриптах

В скриптах оболочки вы можете выполнить аналогичные проверки при помощи утилиты `systemd-detect-virt(1)`. Например:

```
if systemd-detect-virt -q ; then
        echo "Virtualization is used:" `systemd-detect-virt`
else
        echo "No virtualization is used."
fi
```

Эта утилита возвращает код 0 (успех), обнаружив виртуализацию, или ненулевое значение, если виртуализация не выявлена. Кроме того, она выводит идентификатор обнаруженной системы виртуализации (согласно списку выше), если это не было запрещено опцией `-q`. Кроме того, опции `-c` и `-v` позволяют ограничить проверки только механизмами виртуализации на уровне ОС, либо полной виртуализации, соответственно. Подробности см. на странице руководства.

### 19.3 В программах

Информация о виртуализации также представлена на системной шине:

```
$ gdbus call --system --dest org.freedesktop.systemd1 --object-path /org/freedesktop/systemd1 \
> --method org.freedesktop.DBus.Properties.Get org.freedesktop.systemd1.Manager Virtualization
(<'systemd-nspawn'>,)
```

Если виртуализация не выявлена, это свойство содержит пустую строку. Обратите внимание, что некоторые контейнерные системы не могут быть обнаружены напрямую из непривилегированного кода. Именно поэтому мы не стали создавать библиотеку, а воспользовались шиной D-Bus, которая позволяет корректно решить проблему привилегий.

Стоит отметить, что все эти инструменты определяют только «самый внутренний» из задействованных механизмов виртуализации. Если вы используете несколько систем, вложенных друг в друга, вышеописанные инструменты обнаружат только ту, в которой они непосредственно запущены. В частности, если они работают в контейнере, находящемся внутри виртуальной машины, они увидят только контейнер.

## 20 Сокет-активация служб и контейнеров

Сокет-активация — это одна из наиболее интересных возможностей systemd. Еще в первом анонсе нашего проекта мы рассказывали о том, как этот механизм улучшает параллелизацию и отказоустойчивость использующих сокеты служб, а также упрощает формирование зависимостей между службами при загрузке. В данной статье я продолжу рассказ о его возможностях — на этот раз речь пойдет об увеличении числа служб и контейнеров, работающих на одной и той же системе, без повышения потребления ресурсов. Другими словами: как можно увеличить количество клиентских сайтов на хостинговых серверах без затрат на новое оборудование.

### 20.1 Сокет-активация сетевых служб

Начнем с небольшого отступления. Итак, что же такое сокет-активация, и как она работает? На самом деле все довольно просто. systemd создает «слушающие» сокеты (не обязательно IP) от имени вашей службы (которая пока не запущена) и ожидает входящие соединения. Как только в сокет поступает первый запрос, systemd запускает службу и передает ей полученные данные. После обработки запроса, в зависимости от реализации службы, она может продолжать работу, ожидая новых соединений, или завершиться, переложив эту задачу обратно на systemd (который вновь запустит ее при поступлении очередного запроса). При этом, со стороны клиента невозможно отличить,

когда служба запущена, а когда — нет. Сокет постоянно остается открытым для входящих соединений, и все они обрабатываются быстро и корректно.

Такая конфигурация позволяет снизить потребление системных ресурсов: службы работают и потребляют ресурсы только тогда, когда это действительно необходимо. Многие интернет-сайты и службы могут использовать это с выгодой для себя. Например, хостеры веб-сайтов знают, что из огромного количества существующих в Интернете сайтов лишь малая часть получает непрерывный поток запросов. Большинство же сайтов, хотя и должны постоянно оставаться доступными, получают запросы очень редко. Используя сокет-активацию, вы можете воспользоваться этим: разместив множество таких сайтов на одной системе и активируя их службы только при необходимости, вы получаете возможность «оверкоммита»: ваша система будет обслуживать сайтов больше, чем формально позволяют ее ресурсы[66]. Разумеется, увлекаться оверкоммитом не стоит, иначе в моменты пиковой нагрузки ресурсов может действительно не хватить.

С помощью systemd вы без труда можете организовать такую схему. Множество современных сетевых служб уже поддерживают сокет-активацию «из коробки» (а в те, которые пока не поддерживают, ее не так уж и сложно добавить). Реализованный в systemd механизм управления экземплярами служб позволяет подготовить универсальные шаблоны конфигурации служб, и в соответствии с ними для каждого сайта будет запускаться свой экземпляр службы. Кроме того, не стоит забывать, что systemd предоставляет внушительный арсенал механизмов обеспечения безопасности и разграничения доступа, который позволит изолировать клиентские сайты друг от друга (например, службы каждого клиента будут видеть только его собственный домашний каталог, в то время как каталоги всех остальных пользователей будут им недоступны). Итак, в конечном итоге вы получаете надежную и масштабируемую серверную систему, на сравнительно небольших ресурсах которой функционирует множество безопасно изолированных друг от друга служб — и все это реализовано штатными возможностями вашей ОС[67].

Подобные конфигурации уже используются на рабочих серверах ряда компаний. В частности, специалисты из Pantheon используют такую схему для обслуживания масштабируемой инфраструктуры множества сайтов на базе Drupal. (Стоит упомянуть, что заслуга ее внедрения в Pantheon принадлежит Дэвиду Штрауссу. Дэвид, ты крут!)

## 20.2 Сокет-активация контейнеров

Все вышеописанные технологии уже реализованы в ранее вышедших версиях systemd. Если ваш дистрибутив поддерживает systemd, вы можете воспользоваться этими механизмами прямо сейчас. А теперь сделаем шаг вперед: начиная с systemd 197 (и, соответственно, с Fedora 19), мы добавили поддержку сокет-активации *виртуальных контейнеров* с полноценными ОС внутри. И я считаю это действительно важным достижением.

Сокет-активация контейнеров работает следующим образом. Изначально, systemd хост-системы слушает порты от имени контейнера (например, порт SSH, порт веб-сервера и порт сервера СУБД). При поступлении на любой из этих портов входящего запроса, systemd запускает контейнер и передает ему все его сокеты. Внутри контейнера, еще один systemd (init гостевой системы) принимает эти сокеты, после чего вступает в работу вышеописанная схема обычной сокет-активации служб. При этом, SSH-сервер, веб-сервер и СУБД-сервер «видят» только ОС контейнера — хотя они были активированы сокетами, созданными на хосте! Для клиента все эти тонкости скрыты. Таким образом, мы получаем виртуальный контейнер с собственной ОС, активируемый при

[66] Прим. перев.: Стоит отметить, что подобная схема работает только при условии, что для каждого клиентского сайта запускаются отдельные процессы служб, хотя это и происходит в рамках одного хоста. Не очень распространенный в отечественном хостинге вариант: обычно следующей опцией после shared-хостинга (одна служба на всех клиентов) идет VPS (каждому клиенту по виртуальному хосту).

[67] Прим. перев.: В качестве практического примера использования сокет-активации служб systemd в промышленных серверных платформах, можно предложить вот эту статью, наглядно описывающую применение этой и других технологий (мониторинг, использование Journal, ограничение ресурсов и доступа) на примере Node.js.

поступлении входящего сетевого соединения, причем совершенно прозрачно для клиента[68].

Внутри контейнера функционирует полноценная ОС, причем ее дистрибутив не обязательно совпадает с дистрибутивом хост-системы. Например, вы можете установить на хосте Fedora, и запускать на нем несколько контейнеров с Debian. Контейнеры имеют собственные init-системы, собственные SSH-серверы, собственные списки процессов и т.д., но при этом пользуются некоторыми механизмами ОС хоста (например, управлением памятью).

К настоящему моменту сокет-активация контейнеров поддерживается лишь встроенным в systemd простейшим контейнерным менеджером — systemd-nspawn. Мы надеемся, что соответствующая возможность вскоре появится и в libvirt-lxc. А пока, за отсутствие альтернатив, рассмотрим использование этого механизма на примере systemd-nspawn.

Начнем с установки файлов ОС контейнера в выбранный каталог. Детальное рассмотрение этого вопроса выходит далеко за рамки нашего обсуждения, и при том детально рассмотрено во многих статьях и руководствах. Поэтому ограничусь лишь несколькими наиболее важными замечаниями. В частности, команда для установки Fedora будет выглядеть следующим образом:

```
$ yum --releasever=19 --nogpg --installroot=/srv/mycontainer/ --disablerepo='*' \
> --enablerepo=fedora install systemd passwd yum fedora-release vim-minimal
```

а для Debian —

```
$ debootstrap --arch=amd64 unstable /srv/mycontainer/
```

Также см. последние абзацы страницы руководства systemd-nspawn(1). Стоит отметить, что в настоящее время реализация системного аудита в Linux не поддерживает виртуальные контейнеры, и ее включение в ядре хоста вызовет множество ошибок при попытке запустить контейнер. Отключить ее можно добавлением `audit=0` в строку параметров загрузки ядра хоста.

Разумеется, внутри контейнера должен должен быть установлен systemd версии не ниже 197. Установить его и произвести другие необходимые настройки можно при помощи того же systemd-nspawn (пока что в режиме аналога `chroot`, т.е. без параметра `-b`). После этого можно попробовать загрузить ОС контейнера, используя systemd-nspawn уже с параметром `-b`.

Итак, ваш контейнер нормально загружается и работает. Подготовим для него service-файл, при помощи которого systemd сможет запускать и останавливать виртуальное окружение. Для этого, создадим на хост-системе файл `/etc/systemd/system/mycontainer.service` со следующим содержанием:

```
[Unit]
Description=My little container

[Service]
ExecStart=/usr/bin/systemd-nspawn -jbD /srv/mycontainer 3
KillMode=process
```

Теперь мы можем запускать и останавливать эту службу командами `systemctl start` и `stop`. Однако, пока что мы не можем войти в эту систему[69]. Чтобы исправить это упущение, настроим на контейнере SSH-сервер, причем таким образом, что подключение к его порту активировало весь контейнер, а затем активировало сервер, работающий внутри. Начнем с того, что прикажем хосту слушать порт SSH для контейнера. Для этого создадим на хосте файл `/etc/systemd/system/mycontainer.socket`:

[68] Кстати говоря, это еще один аргумент в пользу важности быстрой загрузки для серверных систем.

[69] Прим. перев.: Ручной запуск на хосте соответствующей команды `systemd-nspawn -b` во время работы такой службы просто создаст *еще один контейнер*. Хотя корневой каталог у них один и тот же, пространства имен (например, списки процессов) у каждого будут свои. Впрочем, данная задача может быть решена утилитой `nsenter`, которая должна войти в следующий релиз util-linux (предположительно, 2.23).

```
[Unit]
Description=The SSH socket of my little container

[Socket]
ListenStream=23
```

После того, как мы запустим этот юнит командой `systemctl start`, systemd будет слушать 23-й TCP-порт хоста. В примере выбран именной 23-й, потому что 22-й скорее всего окажется занят SSH-сервером самого хоста. nspawn виртуализует списки процессов и точек монтирования, но не сетевые стеки, поэтому порты хоста и гостей не должны конфликтовать[70].

Пока что systemd, работающий внутри контейнера, не знает, что делать с тем сокетами, которые ему передает systemd хоста. Если вы попробуете подключиться к порту 23, контейнер запустится, но сетевое соединение немедленно будет закрыто, так как данному сокету не соответствует пока никакой сервер. Давайте это исправим!

Настройка сокет-активации службы SSH была детально рассмотрена в одной из предыдущих статей, поэтому ограничусь приведением содержимого необходимых для этого конфигурационных файлов. Файл настроек для сокета (`/srv/mycontainer/etc/systemd/system/sshd.socket`)[71]:

```
[Unit]
Description=SSH Socket for Per-Connection Servers

[Socket]
ListenStream=23
Accept=yes
```

Соответствующий ему файл конфигурации службы (`/srv/mycontainer/etc/systemd/system/sshd@.service`):

```
[Unit]
Description=SSH Per-Connection Server for %I

[Service]
ExecStart=-/usr/sbin/sshd -i
StandardInput=socket
```

После чего, добавим наш сокет в зависимости к цели `sockets.target`, чтобы systemd создавал его автоматически при загрузке контейнера[72]:

```
$ chroot /srv/mycontainer ln -s /etc/systemd/system/sshd.socket \
> /etc/systemd/system/sockets.target.wants/
```

[70] Прим. перев.: Ограниченные возможности виртуализации сети в `systemd-nspawn` значительно затрудняют использование описываемой технологии. Ее практическое применение будет иметь смысл после реализации соответствующей поддержки в lxc-libvirt, либо расширения возможностей nspawn. Впрочем, опытные администраторы легко могут найти обходные пути, например: присваивание хосту дополнительного IP-адреса (безо всякой виртуализации, командой `ip addr add`) и привязка слушающих сокетов к конкретным адресам (в параметре `ListenStream=` сокет-файлов и/или в директиве `ListenAddress` файла `sshd_config` хоста).

[71] Прим. перев.: Обратите внимание на разницу между файлами конфигурации сокетов на хосте и в контейнере. Внутри контейнера используются параметры `Accept=yes` и `StandardInput=socket`, которые не задействованы на хосте. Это обусловлено тем фактом, что внутри контейнера сокет-активация производится в стиле inetd (служба получает только один сокет, причем замкнутый на ее потоки STDIN и STDOUT), в то время как на стороне хоста используется родной стиль активации systemd, при котором запущенному процессу (в данном случае `systemd-nspawn`) просто передаются открытые файловые дескрипторы (`fd`) всех сокетов. Одно из достоинств такого подхода — возможность передавать сразу несколько сокетов, что позволяет активировать внутри контейнера несколько серверов.

[72] Прим. перев.: Возиться вручную с командой `ln` здесь совершенно необязательно. Можно просто добавить в файл `sshd.socket` секцию `[Install]`, содержащую параметр `WantedBy=sockets.target`, после чего добавление правильного симлинка будет выполняться куда более очевидной командой `systemctl --root=/srv/mycontainer enable sshd.socket`.

Собственно, все. После того, как мы запустим на хосте юнит `mycontainer.socket`, systemd начнет прослушивать TCP-порт 23. При подключении к этому порту, systemd запустит контейнер и передаст сокет ему. Внутри контейнера свой systemd, в соответствии с файлом `sshd.socket`, примет этот сокет и запустит для нашего соединения экземпляр `sshd@.service`, что позволит нам залогиниться в контейнер по SSH.

Если нам нужно запуститьунтри контейнера другие службы с сокет-активацией, мы можем добавить в `mycontainer.socket` дополнительные сокеты. Все они будут прослушиваться, обращение к любому из них приведет к активации контейнера, и все эти сокеты будут переданы контейнеру при его активации. Внутри контейнера они будут обработаны соответствии с настройками имеющихся там сокет-юнитов. Те сокеты, для которых соответствующих юнитов не найдется, будут закрыты[73], а те сокеты, которые будут настроены для прослушивания внутри контейнера, но не получены от хоста, будут активированы и доступны изнутри контейнера (а если это сетевые или файловые unix-сокеты, то и извне).

Итак, давайте отступим чуть назад и полюбуемся на результаты наших трудов. Что мы получили в итоге? Возможность настраивать на одном хосте множество контейнеров с полноценными ОС внутри, причем контейнеры запускаются только по запросу, что позволяет снизить потребление системных ресурсов и, соответственно, увеличить количество контейнеров (по сравнению с принудительной их активацией при загрузке хоста).

Разумеется, описанный подход работает только для контейнерной виртуализации, и неприменим к полной, т.е. может быть использован только с технологиями наподобие libvirt-lxc или nspawn, но не с qemu/kvm или xen.

Если вы будете администрировать несколько таких контейнеров, вас наверняка порадует одна из возможностей journal: при запуске на хосте утилиты `journalctl` с ключом `-m`, она автоматически обнаружит журналы гостевых контейнеров и объединит их вывод с выводом журнала хоста[74]. Ловко, не правда ли?

Необходимый минимум технологий для сокет-активации контейнеров, присутствует в systemd, начиная с версии 197. Тем не менее, наша работа в этой области еще не закончена, и в ближайшее время мы планируем доработать некоторые моменты. Например, сейчас, даже если все серверные службы внутри контейнера закончили обработку запросов и завершились, контейнер все равно продолжает функционировать и потреблять ресурсы хоста. Мы просто обязаны реализовать возможность автоматического завершения работы гостевой системы в такой ситуации. Причем у нас уже есть готовые наработки в этой области — мы можем задействовать уже существующую инфраструктуру, обеспечивающую автоматическое засыпание/выключение ноутбука при отсутствии активных задач и пользователей.

Впрочем, пора закругляться, а то статья получается чересчур длинной. Надеюсь, что вы смогли продраться через все эти длинные и скучные рассуждения о виртуализации, сокетах, службах, различных ОС и прочем колдунстве. Также надеюсь, что эта статья станет хорошей отправной точкой при конфигурировании мощных и хорошо масштабируемых серверных систем. За дополнительной информацией обращайтесь к документации или приходите на наш IRC-канал. Спасибо за внимание!

[73] Прим. перев.: Стоит особо отметить, что описанная технология работает только для служб, поддерживающих сокет-активацию в режимах inetd (все классические inetd-службы, кроме встроенных) или systemd (зависят от библиотеки `libsystemd-daemon.so`, либо содержат в исходниках заголовочный файл `sd-daemon.h`). Службы, которые сами открывают себе слушающий сокет и не содержат кода для приема уже открытого сокета, так активировать нельзя. Точнее, в случае с сетевыми и файловыми сокетами, все-таки можно, но первое соединение/сообщение при этом будет «потрачено» на запуск контейнера, и до службы в итоге дойдут только последующие, начина со второго, что сводит выгоды практически к нулю.

[74] Прим. перев.: Этот трюк работает благодаря опции `-j`, которую мы передаем программе `systemd-nspawn` при запуске контейнера (см. файл юнита выше). В соответствии с ней, в каталоге `/var/log/journal` хоста создается символьная ссылку на соответствующий каталог гостя. При этом, так как сообщения от гостей имеют другие machine ID, journalctl хоста не выводит их, если явно не указать `-m`. Подробности см. на страницах руководства systemd-nspawn(1) и journalctl(1).